To appear in *Journal of Automated Software Engineering*, Vol 1, No.1, 1994

A Knowledge-Based Prototyping Environment for Construction of Scientific Modeling Software

Richard M. Keller
Michal Rimon
Aseem Das

Recom Technologies, Inc.
Artificial Intelligence Research Branch
NASA Ames Research Center
M/S 269-2, Moffett Field, CA 94035-1000
Keller@ptolemy.arc.nasa.gov

Abstract

Over the past 30 years, scientific software models have played an increasingly prominent role in the conduct of science. Unfortunately, scientific models can be difficult and time-consuming to implement, and there is little software engineering support specifically available for constructing scientific models. Because these models are not easily specified to scientifically-naive programmers, and because the scientist requires intimate knowledge of the code to conduct experiments, many scientists implement their own models. This coding activity takes valuable time away from their primary scientific mission.

We have developed a knowledge-based software development tool that assists scientists in prototyping scientific models. With a specialized graphical user interface, the scientist constructs a high-level visual specification that captures the essential computational dependencies in the desired model. The system uses its scientific domain knowledge to ensure that the model being built is consistent and coherent. The final product is an executable prototype of a scientific model. Our tool accelerates the model-building process and eliminates the scientist's need to program in a formal language. Furthermore, the models developed with this tool are easier to understand and reuse than typical low-level scientific modeling code. At present, models developed with our system are restricted to those involving non-coupled algebraic and first order ordinary differential equations. Research is ongoing to lessen this restriction and support models with simultaneous equations.

Keywords: scientific modeling, numerical computing, domain-specific software tools

Table of Contents

1 Introduction

1.1 Motivation

The use of scientific software models is pervasive throughout science today. In fields ranging widely from astrophysics to zoology, scientists construct models to analyze data, to validate theories, and to predict a whole variety of phenomena. Despite the critical importance of scientific modeling, there is little software engineering support specifically available to facilitate the model-building process. We believe that the development and introduction of intelligent tools to support scientific model-building activities can have a significant impact on scientific productivity and on the quality of scientific models produced.

Construction of scientific modeling software is similar to construction of conventional software in many respects. However, there are several characteristics of the scientific model-building domain that make it somewhat unique and worthy of special attention. First, in this domain, the software actually represents a concrete manifestation of a new scientific theory -- a genuine scientific artifact that should be shared and validated within the scientific community. Although the new theory is typically described in a scientific journal article, potentially important scientific details and modeling assumptions may be skimmed over or omitted entirely in the written description. The complete details of the theory can only be found in the code itself. Unfortunately, a thorough examination of the code is often necessary to recover this information. Such an examination can require considerable time and effort, as the scientific details are difficult to discern at the code level. This impenetrability of modeling code prevents colleagues from attaining a thorough understanding of a new theory, and serves as a barrier to reuse of scientific modeling code. Scientists are legitimately hesitant to reuse unfamiliar code because they know that conflicting assumptions buried in a colleague's code could invalidate their own scientific results. In sum, there is perhaps a greater than typical institutional burden on scientists to produce code that is understandable and reusable.

Another unique characteristic of modeling software is that the scientific end-user is often directly involved in the design and implementation of the code. This direct involvement is the result of several factors. First, programming resources are typically scarce in scientific labs, and researchers may have no choice but to do their own programming. Even when programmers are available, it may be difficult and time-consuming for the scientist to convey the necessary scientific details unless the programmer is well-versed in the scientific discipline. Finally, once the initial code is complete, the scientist must run experiments to verify the model. This may involve considerable changes to the code and may be awkward and inconvenient unless the scientist is intimately familiar with the implementation. A combination of these factors may force a scientist unwillingly into the programmer's role. In all

likelihood, the scientist's limited time would be better spent conducting scientific activities than programming.

Finally, scientific coding presents an unusual combination of special technical challenges to a programmer. Frequently, scientific computing is computationally intense, and the programmer may need to design special algorithms, optimizations, and/or data structures to combat time or space complexity. Often these scientific computations are numerically sensitive and require application of sophisticated numerical analysis techniques. In coding numerical applications, the programmer typically must access large, complex numerical subroutine libraries and access large numerical datasets to piece together a solution to a numerical computation. When solving differential equations, the programmer must design and manage a computational grid structure that appropriately discretizes the continuous parameters in a model. Furthermore, the programmer faces significant bookkeeping overhead required to keep track of scientific units and to maintain the integrity of a numeric computation throughout. Lastly, there is the pervasive problem of maintaining high-quality documentation for the scientific modeling code. Individually, none of these requirements is unique, but in combination, they characterize a suite of special skills required by programmers in scientific software development environments.

1.2 The SIGMA System

We have developed the SIGMA system (Scientists' Intelligent Graphical Modeling Assistant) at NASA Ames Research Center to begin addressing some of the scientific model-building community's diverse software development needs. SIGMA is an interactive knowledge-based tool that helps scientists prototype scientific models. Through an interactive dialog with the scientist, the system develops a complete, coherent, and executable prototype of a scientific model. To facilitate this process, SIGMA provides scientist-users with a high-level visual specification language in which to express scientific models. Within this language, users can specify how scientific quantities of interest can be computed by applying equations or subroutines to known input data. The terms in SIGMA's language reference familiar domain-specific scientific constructs (e.g., specific physical quantities, scientific equations, and datasets). The language is at a sufficiently high level that it omits most implementation details involving data structures and fine-grain control. SIGMA's reasoning components infer these details by referencing extensive domain knowledge and by carrying out clarifying interactions with the user. Once the specification is complete, SIGMA can execute the scientific model and provide visual feedback on the results by displaying data plots.

We chose to develop SIGMA as a specialized tool for the scientific modeling domain, rather than provide users with a general-purpose, broad spectrum software engineering tool. We felt that a

general-purpose tool would have been provided more power than required for the task, and would have proved too open-ended and unfocused for scientists. Scientific models constitute a restricted class of programs, and there is special expertise associated with developing these programs that can be exploited to build a customized tool to suit the task. We found that there were significant benefits to developing a specialized tool. By restricting the domain, we were able to employ extensive knowledge about the modeling task and the user's design goals. As a result, SIGMA can accept less complete software specifications than would be required by a general-purpose software design tool. Low-level details omitted by the user can be supplied by the system based on its domain knowledge. Furthermore, using this domain knowledge, the system can provide more intelligent assistance to users during the software design process. Although the range of programs that can be specified using our system is considerably more restrictive than with a general-purpose software design tool, SIGMA is also easier and more intuitive for scientists to use. This kind of tradeoff between generality and ease of use is the hallmark of domain-specific software design systems (Barstow, 1985; Keller, 1992).

SIGMA was designed as a rapid-prototyping tool for building scientific models. We believe that model prototyping is an area where automated techniques can provide great leverage, and where knowledge-based approaches can make a significant impact on the scientific community. We made a strategic decision to focus on providing knowledge-based support for model construction, rather than to expend our limited resources generating efficient code. In fact, SIGMA does not produce actual code. Instead, a model is represented in SIGMA as a set of explicit data structures, and execution is handled by a special-purpose interpreter. We find that this is adequate for exploratory prototyping activities and simple modeling tasks, but ultimately a model must be compiled into efficient code if it is to be used routinely and involves any significant amount of computation or data. Nevertheless, because scientific model-building is an inherently exploratory activity, we believe users will find value in a flexible prototyping tool, even if it does not produce optimized code. Much of the effort in model-building goes into the initial conceptualization and design process, and SIGMA provides significant help with these activities. After designing a prototype using SIGMA and running some test cases to validate the model, users may find it less time-consuming to produce efficient code in a conventional high-level language of their choice. The prototype model can be used as a basis for implementation of production-quality code. Of course, the proper solution to the code-generation problem is to write a model compiler that produces high-quality, optimized code from SIGMA's internal structures. We believe that model compilation is relatively straightforward, and could be handled using conventional compiler and knowledge-based synthesis techniques. This is an area for future work.

1.3 A perspective on model-building

Before learning how SIGMA assists in developing prototypes of scientific models, it seems crucial to have a firm understanding of exactly what constitutes a model. The conventional view is that a scientific model can be expressed as a set of numeric equations, and that the modeling code implements these equations. From our perspective as researchers interested in automatic generation of modeling software, this view is too narrow. Although numeric equations may form the core of a model, these equations must be annotated with a considerable amount of information before they can be turned into executable modeling code. As syntactic entities, equations express abstract numeric relationships among equation variables. However, in order to properly implement a set of equations in the context of a specific modeling situation, it is critical to understand the correspondence between the equation variables and the attributes of the physical situation being modeled. Scientists understand these relationships and take them for granted. When a scientist implements a scientific model, he or she implicitly uses this knowledge -- plus a great deal of other background scientific knowledge -- to build a semantically correct implementation of the equations. If we are attempting to build a software assistant that can interact intelligently and synergistically with a scientist to create modeling software, we must provide the system with the background knowledge that the scientific user takes for granted.

Our approach is to acquire this background knowledge from scientists, encode it using a uniform object-oriented representation, store it in a knowledge base, and make it available to our system. SIGMA uses this knowledge to support its inferencing, retrieval, and user interface requirements. The knowledge base provides a common language and vocabulary in which to express aspects of a scientist's theory that must be referenced during the model construction process. We encode not only scientific facts and equations, but also the details of the experimental situation posited by the investigator. This contextual information is essential to our goal of providing the system with a basis for understanding the semantics of numerical modeling equations. As we will see, although the end product of the scientific modeling process is a piece of numerically-intensive software, the process of creating that software involves considerable symbolic reasoning.

To summarize, our approach with SIGMA has been to extend the notion of a scientific model beyond a set of equations so that it explicitly encompasses the user's internal model of the objects, attributes, and relations that describe the experimental situation. By making the internal model explicit, we are able to exploit this model in concert with other background knowledge to guide the model-building process.

In the balance of this paper, we describe how SIGMA uses domain knowledge to assist the scientist-user in constructing an executable model prototype. In Section 2, we describe the design requirements for

the SIGMA system. Section 3 follows with a guided tour of the SIGMA system. Section 4 describes SIGMA's equation representation, which forms the foundation for SIGMA's intelligent assistance capability. Section 5 describes SIGMA's knowledge base, which provides the essential background information that is used in interpreting and applying equations. Section 6 revisits the guided tour once again and elaborates on some of the mechanisms used to ensure the construction of scientifically coherent models. Finally, Section 7 reviews related research and Section 8 discusses current status and future directions.

2 Design for a scientific software assistant

In designing SIGMA, our goal was to provide a rapid-prototyping environment that would allow scientists to formulate, implement, test, and modify models. Currently, the scientist accomplishes these model-building steps with little or no automated assistance. When a scientist wishes to formulate a model, he or she might begin by sketching out some of the relevant theory in terms of equations, diagrams, and perhaps some narrative text. All of this information is accumulated in a scientific notebook, which serves as a kind of informal medium for specifying the scientific model. Next, the scientist turns the informal specification into code. Of course, this translation process is non-trivial; Much of the information necessary to transform the notebook sketch into an executable scientific model is implicit scientific background knowledge. The scientist takes this knowledge for granted, and it never appears in the notebook. When the modeling code is complete, the scientist executes the model, analyzes the results, and debugs the model as necessary. The entire model-building process iterates until the scientist has a satisfactory answer to his or her original scientific inquiry.

The primary challenge we faced in designing SIGMA was to come up with a comfortable specification medium for scientists designing models -- a medium as simple, informal, and exploratory as the scientists' notebook, yet one that enforced enough structure that it could be mechanically augmented and transformed into a complete, executable model. We particularly wanted to avoid requiring scientists to tediously recapitulate implicit common-sense background science knowledge that would be required to translate the informal specification into code. This design goal was reinforced by our collaborators' express desire to communicate modeling problems to SIGMA at a level of detail that would be sufficient for graduate students to understand and code. If they were required to convey a lot of basic scientific knowledge as part of building a model, our collaborators indicated they would rather code the model themselves in Fortran.

Aside from minimizing the need to specify background knowledge, we wanted the specification to be simple and intuitive so that scientists could quickly grasp the scientific essence of any specified model.

In particular, we wanted the specification to use a vocabulary familiar to scientists -- one based on meaningful scientific constructs from the modeling domain. We felt this requirement would increase the likelihood that the specifications produced would be understandable and reusable across different groups of scientific users. We also wanted the specifications to be at a sufficiently high level so that most low-level programming details (e.g., fine-grained control, data structure management, database access) could be omitted. We felt this requirement would decrease the level of programming sophistication required to implement a useful scientific model.

These system design criteria -- exploratory yet structured environment, minimal specification of background science knowledge, scientific intuitiveness of specifications, and ease of use -- guided us in the development of SIGMA. To satisfy the exploratory requirement, we implemented SIGMA as a CAD-like environment featuring a direct manipulation graphical user interface. To satisfy the minimal knowledge specification requirement, we shifted the knowledge burden from the user to the system by developing an extensive scientific knowledge base containing general scientific computing knowledge and domain specific knowledge. This knowledge is used to augment and formalize the user's informal specifications. To satisfy the intuitiveness and ease of use criteria, we utilized data flow diagrams as the basic structure for specifying scientific models. These diagrams specify the basic scientific dependencies between quantities computed in a model.

At the onset of our work, we decided to restrict the class of scientific models we would support with the first trial version of our system. This class includes only models describable as sets of non-coupled algebraic equations and first order ordinary differential equations. Due to their lack of simultaneity, models containing non-coupled equations are easy to visualize using data flow diagrams. Also, models with non-coupled algebraic and first order ordinary differential equations require less sophisticated solution procedures than models with simultaneous or higher-order differential equations. Although these restrictions exclude important scientific modeling problems (notably, those involving partial differential equations), SIGMA can still handle many problems of practical interest. The success of our initial approach has led us to investigate methods of extending the scope of our system to incorporate simultaneous and higher order differential equations in future versions of SIGMA.

3 A guided tour of SIGMA

In this section, we provide a brief guided tour of SIGMA and illustrate how models can be built using the system. The purpose of this tour is to introduce SIGMA's capabilities, and to set the stage for a more in-depth discussion of the representation and inferencing techniques that underlie these capabilities. As background for this guided tour, we first describe a particular scientific experiment that will be used

to illustrate SIGMA's model-building interaction. Next, we take the reader through a sequence of model-building steps in SIGMA, accompanying each step with illustrative screen images produced by the system's graphical user interface. Finally, we highlight the main technical challenge that we faced in designing SIGMA to achieve the behavior illustrated in the guided tour.

3.1 Titan Atmospheric Modeling

To demonstrate how SIGMA works, we will illustrate with an example from our work with planetary scientists at NASA's Ames Research Center. We will demonstrate the reconstruction within SIGMA of a small fragment of a planetary atmospheric model that was originally developed (in Fortran) to investigate the thermal properties of Saturn's moon Titan (Lindal, Wood, Hotz, Sweetman, Eshelman, & Tyler, 1983; McKay, Pollack, & Courtin, 1989). The scientists' goal in building this model fragment was to calculate a profile of Titan' s atmosphere that describes the density, pressure, and temperature at various altitudes above its surface (see Figure 1). The major source of experimental data

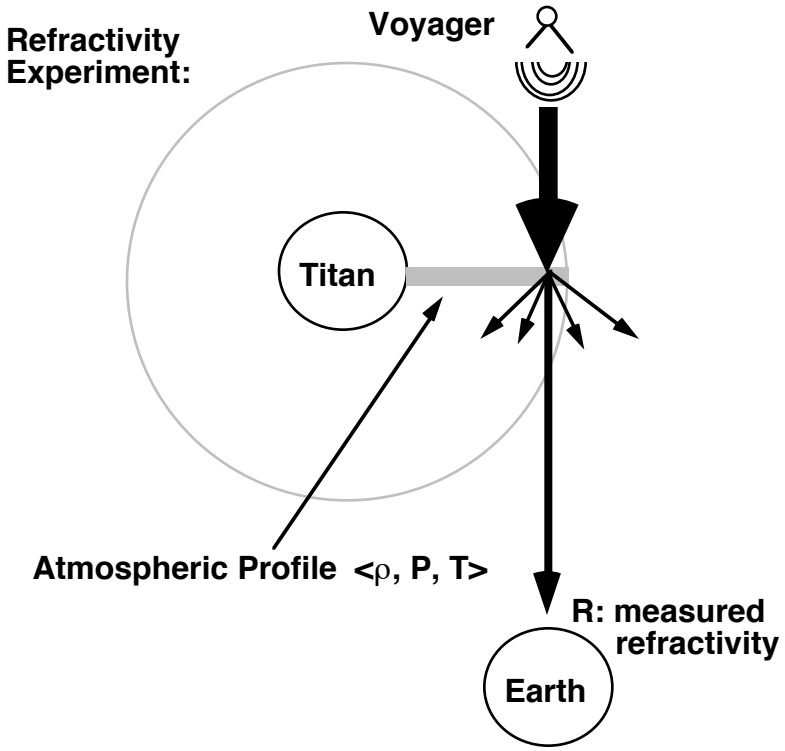

Figure 1: The Voyager refractivity experiment. Based on refractivity data collected during the Voyager experiment, the scientist's modeling goal is to develop an atmospheric profile. The profile must specify density (ρ), pressure (P), and temperature (T) as a function of altitude.

relevant to Titan modeling was generated by the Voyager-I flyby of Titan in November 1980. As Voyager-I reached the far side of Titan, it sent back radio signals that passed through Titan' s atmosphere and then on to receiving stations on Earth. Due to the density of gases in the atmosphere, the radio waves were refracted slightly as they passed through the atmosphere, resulting in a diminished signal picked up on Earth. The amount of refraction was measured at different altitudes above the surface by panning the radio signal sent from Voyager and sampling at discrete points. This refractivity data serves as a starting point for inducing the desired atmospheric profile.

Although we focus on Titan atmospheric modeling in this paper, we have also done extensive work in another scientific domain to illustrate the generality of our approach (Dungan & Keller, 1991). In particular, we have used SIGMA to develop large portions of a forest ecosystem model that tracks the carbon, water, and nitrogen cycles in a forest ecosystem (Running & Coughlan, 1988).

3.2 Steps in Model-building

This section describes five general steps involved in implementing a model with SIGMA:

1. Establish the modeling scope;
2. Specify a goal quantity;
3. Construct the model;
4. Execute the model;
5. Revise the model.

We illustrate these general steps in terms of the Titan atmospheric modeling problem. The specific goal of the illustrated modeling activity is to specify the density portion of the Titan atmospheric profile.

3.2.1 Step 1: Establish the modeling scope

As the first step in building a model with SIGMA, the user must make a number of choices to define the scope of the modeling session and set the context for the modeling activity that will follow. In particular, the user selects from a fixed set of parameterized, **experiment scenarios**. These experiment scenarios are predefined by a knowledge engineer in collaboration with a domain scientist. Each experiment scenario describes the experimental setup for a different modeling situation using an object-oriented representation. This representation includes objects, attributes, and values that a scientific expert feels are relevant to describing the particular experimental situation. Each object in the scenario is defined in SIGMA's scientific knowledge base, and stores both a set of quantities associated with the object and a set of links to related objects. After the user selects an experiment scenario, SIGMA

instantiates the scenario's domain objects and links them together according to the scenario definition. After setting up the domain object instances, the system checks the experiment scenario definition to determine whether initial values for any quantities have been specified. If so, SIGMA loads these initial values into the specified quantity attributes of the instantiated domain objects. An example will clarify the role of the experiment scenario.

In our example, the user has selected the "Lindal[1] Scenario" from the TGM (Titan Greenhouse Model) modeling domain (see Figure 2). This scenario is parameterized by two items: the set of atmospheric constituents and the number of Voyager refractivity data points to be analyzed. In this case, the user has chosen to include gaseous hydrogen and nitrogen as part of the Titan atmosphere, and wishes to analyze a total of 106 data points. Figure 3 illustrates part of the structure of interrelated objects and quantities that results from instantiating this parameterized scenario. The arrows in the figure represent links between the objects. This scenario represents the Voyager-I flyby experiment described in Section 3.1. In the scenario, the atmosphere of Titan is modeled by a sequence of spatially adjacent **atmospheric-parcel** objects, which are stored as an array of links in the "parcels" slot of the **Titan** object. Each atmospheric-parcel represents a mixture of gases at a specified altitude above Titan's surface. Each of these gases is described by a **constituent** object. The atmospheric-parcel and both constituents are being irradiated by a common radiation-source: the **Voyager-signal** object. The **radiation-interaction** objects in the figure represent the actual collision of the radiation with the irradiated material. (There are three separate radiation-interaction objects because the collision with each constituent is modelled separately from the collision with the atmospheric-parcel.) Quantities that are a function of both the radiation source and the irradiated material are stored in the radiation-interaction objects. For example, the amount of **refractivity** caused when the Voyager radio signal penetrates the atmospheric parcel is a function of both the wavelength of the Voyager signal and the properties of the gaseous material in the parcel. Note that in this scenario, the refractivity value is known *a priori* based on earthbound observations. Thus, the value of the refractivity attribute associated with Radiation-Interaction-14 has been initialized. Other values are also known based on background domain knowledge in SIGMA's knowledge base. For example, the molecular weights of hydrogen and nitrogen are known. In essence, the structure in Figure 3 is a formal representation of the experimental situation depicted in Figure 1.

[1]The "Lindal Scenario" is named after G. Lindal, who was the principal investigator in charge of the Titan-Voyager experiment.

Figure 2: Parameters for instantiation of "Lindal" experiment scenario

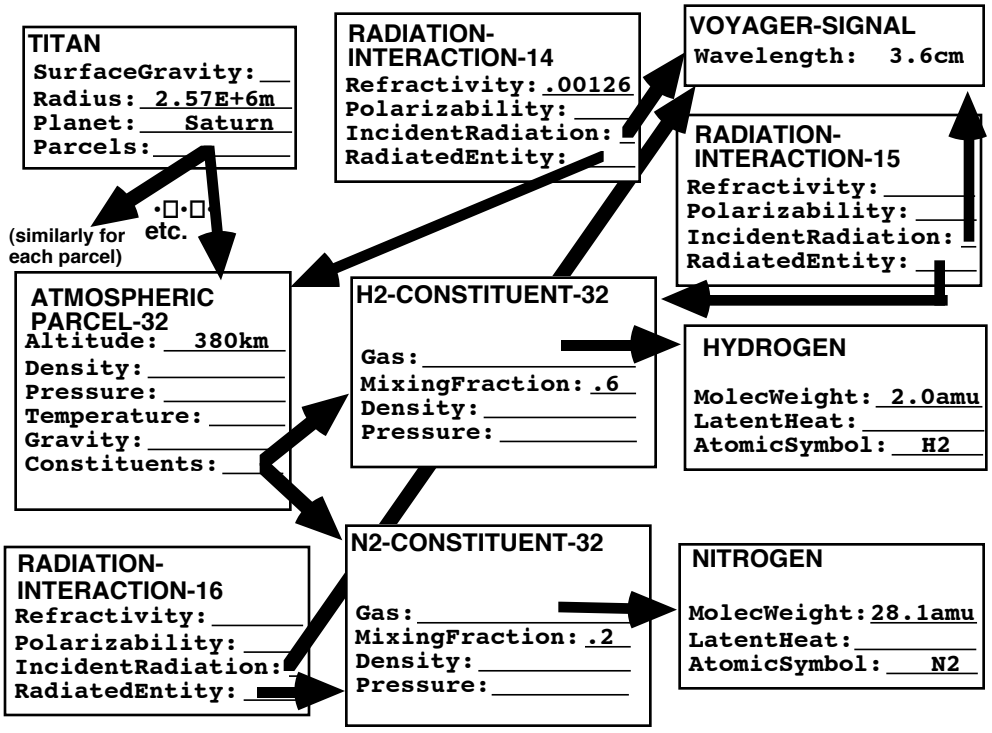

Figure 3: Instantiated "Lindal" experiment scenario.

The fully instantiated Lindal scenario actually contains an array of structures isomorphic to the one depicted in Figure 3. There is one structure associated with each atmospheric-parcel link stored on the parcels slot of the Titan object (i.e., one structure for each altitude level at which a refractivity data point is to be analyzed). This array structure corresponds to what is referred to conventionally as a discretized computational "grid". In our case, the grid is one-dimensional, but often, grids are higher-dimensional.

A more complete description of the Lindal Scenario -- including a full specification of the objects, quantities, and interconnections -- can be found in Appendix 2.

Having introduced the notion of an experiment scenario, we can now describe the aim of a SIGMA modeling session more precisely in terms of the scenario. The aim of the modeling activity is to specify a computational method for determining one or more of the unknown quantities in the experiment scenario. We call these quantities "goal" quantities because they are in some sense goals of the modeling activity.

3.2.2 Step 2: Specify a goal quantity

To specify a goal quantity, the user selects from a list containing all quantities associated with any of the objects in the experiment scenario. Because a given quantity may be associated with several objects in the scenario (e.g., the mass of Titan, the mass of an atmospheric parcel, the mass of hydrogen, etc.), the user must also identify an object in order to uniquely specify the goal quantity (see Figure 4). In addition, the user must specify the scope of the quantity in cases where it recurs within an array-structured computational grid. For example, as discussed above, the Lindal Scenario (Figure 3) includes a one-dimensional grid array of atmospheric parcel objects, representing the parcels at various altitudes above Titan's surface. If the user selects the density of an atmospheric parcel as the goal quantity, he or she must also specify whether their intention is to compute all of the density quantities on the grid, a subset of these quantities, or just a single quantity. Once all of the goal quantity selection decisions have been made by the user, SIGMA displays a single node representing the entire scope of the selected goal quantity as the starting point for subsequent model-construction activity (see Figure 5).

Figure 4: Selection of the goal quantity.

Figure 5: Initial model construction window contains a single node representing the goal quantity.

3.2.3 Step 3: Construct the model

After the goal quantity has been established, the next step is to formulate a model capable of computing the quantity. The construction of the model involves assembling a sequence of numerical transforms (i.e., equations and/or subroutines) that can be applied to known quantities in the experiment scenario in order to compute the desired goal quantity. Within SIGMA, a model is represented using a data flow structure that describes how the goal quantity is computed from other quantities via the application of transforms. Although the model execution process starts with known quantities and calculates unknowns from them, the model construction process is conducted in the reverse order, working backward from the unknown goal quantity toward the known quantities.

To begin the construction process, the user clicks on the right arrow button of the density quantity node in Figure 5, indicating his or her desire to extend the initial model (which consists of just a single node) to the right. By convention, quantities (i.e. data) flow from right to left in the data flow diagram. Thus, extending the model to the right of a quantity Q involves finding a computation to compute Q, whereas extending the model to the left involves using Q to derive some other quantity.

To identify possible extensions to the model, SIGMA searches through its knowledge base to find all numerical transforms that can be applied to compute the density quantity from other quantities. Numerical transforms consist of either explicit scientific equations, which are created within SIGMA using an equation entry tool, or foreign "black box" subroutines, which are externally created and imported into SIGMA. Once the system has identified an initial set of candidate transforms, the set is filtered to remove any transform that is not semantically meaningful to apply in the current situation. Specifically, SIGMA filters out any transform that is 1) invalid to apply in the context of the current experiment scenario, or 2) inconsistent with the model fragment constructed thus far. In Figure 6, the transforms that remain after this filtering process appear in the list entitled "Applicable Tranforms". For informative purposes, transforms that were filtered out are displayed in the "Near Miss Transforms" list. This knowledge-based transform-filtering process is a key feature of SIGMA's functionality, and is more fully described in Section 6.

The user must now select one of the applicable transforms to include in the model. The user's selection is based on his or her scientific expertise; SIGMA provides information about each transform (as shown in the equation description and bibliographic citation windows displayed in Figures 7 and 8), but does not make a choice for the user. In our example, the user chooses to apply the equation labeled "Density Computation". SIGMA applies the equation and extends the data flow graph to the right as shown in Figure 9. The node with the thicker border represents the selected transform, and the polarizability and refractivity nodes to its right represent the input quantities required by the transform to compute the density quantity. Although it is not visible in the black and white reproduction of SIGMA's color

interface, each quantity node in Figure 9 is colored to indicate the value status of the node (see Figure 2.) Yellow (labeled "Y" in the figures) indicates that the value of the associated quantity is *known*; White (labeled "W") indicates that the value of the quantity is *computable* from other quantities; Red (labeled "R") indicates that the value of the quantity is *unknown*. In Figure 9, the refractivity node is yellow because the value of the refractivity is known in the initial experiment scenario (Figure 3). The polarizability node is red because there is no value known for this quantity. The coloring of the density node switches from its initial red color in Figure 5 to white in Figure 9 after the data flow diagram is extended to the right using the density calculation equation; The density value is now computable from the polarizability and refractivity values.

To proceed in developing the model further, the user has two options: manually input a value for polarizability (via the "Input" button on the node) or compute its value by extending the model further using the right arrow button. In our case, the user extends the model by selecting the "Polarizability of a Gaseous Mixture" equation (Figure 10) from the applicable transforms presented by the system. SIGMA augments the data flow display to reflect this selection as indicated in Figure 11. Note that there are two sets of quantity nodes required as inputs to this equation because the equation is a summation over the two atmospheric constituents. Since the user selected nitrogen and hydrogen constituents in the Lindal Scenario setup (see Figure 2), a mixing ratio and a polarizability is required for each constituent.

Figure 6: Transform selection options.

Figure 7: Description of equation labeled "Density Computation".

Figure 8: Bibliographic citation for "Density Computation" equation.

Figure 9: Model construction window after Density Computation has been applied. A node label "Y" indicates that the value of the associated quantity is known; a node label "W" indicates that the value of the quantity is computable; a node label "R" indicates that the value of the quantity is unknown.

Figure 10: Description of equation labeled "Polarizability of a Gaseous Mixture".

Proceeding further, the user continues to augment the model by repeatedly expanding the computation beneath unknown (i.e., red-colored) quantity nodes in the data flow diagram, using the right arrow button to select a transform to apply. The user can also choose to input a value for an unknown quantity at any time. For example, the user might enter the mixing ratios associated with the nitrogen and hydrogen constituents in the "Polarizability of a Gaseous Mixture" equation. The model construction process terminates when all unknown nodes are eliminated. In other words, the model is complete when the value of all quantity nodes in the diagram are either known (as a result of being initialized in the experiment scenario or input by the user) or computable from these known values. The user can expand the model in any order, and can undo previous decisions by re-clicking on the right arrow button of a an already-expanded quantity node. In this case, SIGMA will permit the user to choose an alternative transform for computing the quantity, and will excise the data flow graph section that relates to the old transform. The user can now continue model-building using the new transform. Figure 12 illustrates the final completion of the partial model illustrated in Figure 11.

Figure 11: Model construction window after application of equation labeled "Polarizability of a Gaseous Mixture".

Figure 12: Completion of model to compute density goal quantity.

3.2.4 Step 4: Execute the model

After the model is complete, the user can click the "Compute" button on any quantity node to compute its value. SIGMA's graphical user interface will graphically simulate execution of the data flow diagram to help the user visualize the computation.

Basic execution is relatively straightforward, and is typical of data flow execution schemes (Davis & Keller, 1982). To compute a given quantity, the system executes each transform in the data flow subgraph rooted at the quantity node. Execution proceeds in order from the fringe of the subgraph back to the root. Before a transform can be executed, all its inputs must be computed. The execution sequencing is handled by a queuing mechanism. Any transform with all of its input values known is placed on a queue to be fired. SIGMA's model-interpreter selects and executes transforms one at a time from this queue. When executing an equation, the interpreter converts all inputs to a common scientific unit system, and then performs the mathematical operations specified in the equation formula. If the equation is a differential equation, the interpreter calls a differential equation solver to solve for the output quantity. For this purpose, SIGMA makes a foreign function call to the LSODA procedure in ODEPACK (Hindmarsh, 1983). ODEPACK is a public domain, Fortran-based package of numerical routines available through the NETLIB server. LSODA will solve the differential equation and return a solution to SIGMA.[2] When executing a subroutine, the interpreter converts all inputs to the scientific units expected by the subroutine's arguments, and then executes the subroutine via a foreign function call. Once a transform is executed and its output quantity is computed, additional transforms that require the new quantity may queue themselves for firing. The interpreter continually selects and executes transforms until the desired quantity has been computed. After the quantity has been computed, the user can click on the node's "Info" button to inspect or plot the computed value (see Figure 13 and 14).

Aside from this simple forward-chaining execution scheme, SIGMA's interpreter supports basic conditional and iterative control, as well. Data flow control nodes can be inserted into the data flow diagram using the 'split' and 'join' buttons at the top left of the model-construction window. These constructs split and join data flow paths based on the result of some logical test during execution. These constructs allow the user to build forks and loops in the data flow path.

[2] SIGMA must provide certain auxiliary information to LSODA, such as the boundary conditions, the interpolation method, and other solution method parameters. This information is provided by the user prior to execution, as a part of the interaction in which the differential equation is selected.

Figure 13: Display of computed values for density goal quantity.

Figure 14: Plot of values for density goal quantity.

3.2.5 Step 5: Revise the model

If the model results do not match the user's expectations, or if the user wants to perform sensitivity analyses, he or she can modify the model in a variety of ways and re-execute. For instance, the user could change the value of any input quantity (e.g., the mixing ratios of hydrogen and nitrogen in Titan's atmosphere). Or the user might choose a different transform to compute a particular quantity in the model (e.g., by selecting the "Ideal Gas Law" rather than the "Density Calculation" in Figure 6).

At any point in the model-building process, the user can save the current model in the user's personal model library. At the beginning of a subsequent modeling session, the user can request that SIGMA automatically restore the previously-saved model. The user can now continue with the model-building process as before. This feature enables users to interrupt their work and resume at a later date without requiring them to repeat previous model-building steps from scratch.

3.3 Technical challenge: scientific coherence

The primary technical challenge we faced in developing SIGMA was to design a set of representation and inference mechanisms to ensure that models constructed with the system would be *scientifically coherent*. The evaluation of a model's scientific coherence has both a local and a global aspect. Locally, this evaluation involves determining that each transform within the model is appropriately applied. In other words, the scientific preconditions for applying each transform must be satisfied. Globally, the evaluation of scientific coherence involves ensuring that the model transforms are consistent with each other and that they fit together properly to achieve the desired result. We wanted to provide SIGMA with the ability to evaluate scientific coherence for a couple of reasons. First, we wanted SIGMA to present a set of intelligent transform application choices to the user. Otherwise, the user might mistakenly choose to apply an inappropriate transform, and this would lead SIGMA to produce a model that was fundamentally ill-formed.[3] Second, we wanted SIGMA to dynamically monitor the overall consistency of the model being constructed, and prevent inconsistencies from being unintentionally introduced by the user. These two abilities greatly increase SIGMA's utility as a scientist's assistant and are crucial to its success.

Although maintaining scientific coherency is second nature to a domain scientist, automating this process requires access to a great deal of semantic knowledge about transforms, domain objects, and the experiment scenario. In the following sections, we will discuss the representation and inference

[3]We make no attempt to ensure that the models produced with SIGMA are "correct" models of the phenomenon being studied -- only that they are well-formed.

mechanisms that SIGMA employs to evaluate and enforce scientific coherency. Then we will revisit the SIGMA guided tour to explain how these mechanisms work together to maintain scientific coherency during model-building. We start with SIGMA's equation representation, which forms the foundation for much of SIGMA's intelligent functionality.

4 Scientific equation representation

Although the mathematical relationship expressed by any given scientific equation is clear from its syntax, the syntactic representation fails to convey a variety of semantic information that is essential to a real understanding of the equation. Much of the meaning inherent in an equation is bound up in the interpretation of its symbols. Scientists in the same field tend to understand the meaning of equations due to their shared understanding of syntactic conventions and their common scientific background. But scientists from a different domain have little hope of understanding a particular modeling equation without further explanation and annotation.

Consider the following equation:

$$F = \frac{Gm_1m_2}{r^2}. \tag{1}$$

Physicists will recognize this equation as the Gravitational Force Equation, which describes the gravitational force, F, between two bodies with mass m_1 and m_2, respectively. A physicist knows that r in this equation represents the separation distance between the centers of mass of the two bodies, and G represents the value of the Universal Gravitational Constant. The force is directly proportional to the product of the masses and inversely proportional to the square of the separation distance. The diagram that forms the basis for a proper interpretation of Equation 1 is illustrated in Figure 15. This diagram is a description of the generalized physical situation in which the equation applies. The semantics of the symbols in Equation 1 are intimately related to this situation. The symbols do not make sense in a vacuum -- they only make sense with reference to objects and quantities in the diagram.

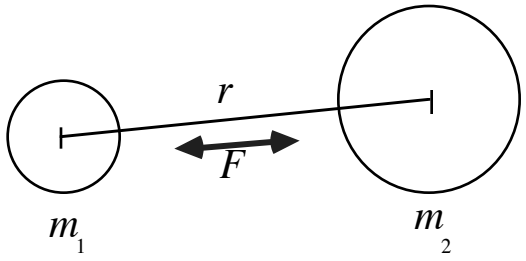

Figure 15: Interpreting the Gravitational Force Equation.

Evidently, a physicist brings a great deal of information to bear in understanding an equation based on his or her training and background knowledge. This information goes far beyond the syntactic symbols in the equation. Although non-physicists may understand the mathematics behind the Gravitational Force Equation, they will be at a loss to explain the real meaning of the equation in terms of the physics of interacting bodies. To understand further, they must be given access to semantic information of the form described above.

Just as a non-physicist requires additional information to understand the meaning behind a physics equation, a computer-based modeling assistant requires such information. In particular, SIGMA requires a semantic interpretation for each equation in order to decide which equations can be applied legitimately at any given point in the model construction process, and to ensure that the equations in a model fit together in a consistent, coherent fashion.

To capture the type of semantic information outlined above, we designed an equation representation similar to those developed for reasoning about lumped-parameter models of physical systems (Nayak, 1992). Each equation consists of a syntactic formula annotated with additional information that constrains the symbols within the formula, and associates the symbols with quantity attributes of objects in the scientific domain. To illustrate, Figure 16 graphically depicts our representation for the Gravitational Force Equation. Each box in the figure represents a generalized class of scientific domain objects, each of which has a defined set of slots (i.e., attributes). These slots store either quantities or links to other objects. (The object links are displayed as thick arrows in the Figure 16.) Each symbol in the formula is connected to a quantity slot associated with one of these objects. (This connection between formula symbols and quantity slots is represented by a thin arrow in Figure 16.) For example, the symbol m_1 refers to the mass of one of the bodies, which is a type of physical-entity object. Similarly, the symbol m_2 is the mass of another physical-entity, which is distinct from the first. These two physical entities are related to one another through a force-field object using the first-body/second-body links. The symbols *r* and *F* are the length and force associated with this force-field object[4]. Finally, the symbol *G* is linked to a constant that stores the value of the Universal Gravitational Constant.

[4]If two different force-field objects were necessary to describe the equation, they would be depicted separately. Otherwise, symbols depicted as pointing to slots in the same object are constrained to refer to the same object.

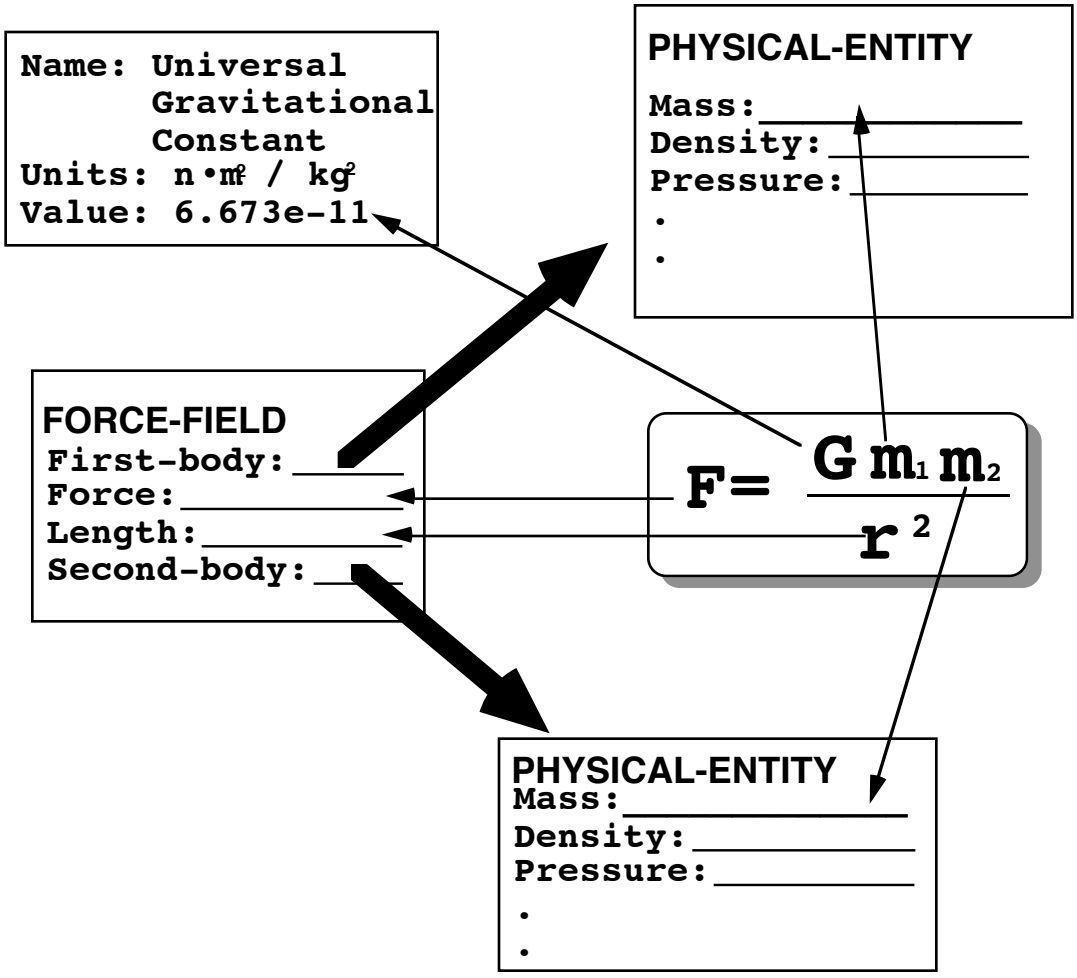

Figure 16: Representation for Gravitational Force Equation

This graphical representation specifies a general applicability pattern for the equation. The pattern is general because it specifies the object classes involved, rather than any specific instances. The objects and constraints in the pattern must be matched against actual instances to determine if the applicability conditions are met in a specific domain situation. The applicability conditions can be expressed formally using predicate calculus notation. Given a set of three instantiated objects, pe1, pe2, and ff, the applicability conditions for the Gravitational Force Equation are as follows:

physical-entity(pe1)

^ *physical-entity(pe2)*

^ *force-field(ff)*

^ *first-body(ff,pe1)*

^ *second-body(ff,pe2)*

^ *not-equal(pe1,pe2).*

A further constraint (not illustrated by the graphical representation) must be stated to ensure that the length of the force field is the same as the separation distance between the physical entities:

^ *equal(length(ff), center-distance(pe1, pe2)),*

where the function *length* retrieves the length attribute of the force-field, and the function *center-distance* computes the distance between the coordinates of the centers of mass of the two physical entities.

Further details on the representation for the Gravitational Force Equation are given in Appendix Section A1.2.

SIGMA's representation for subroutines is almost identical to its representation for equations, with the following exceptions. First, the syntactic "formula" associated with a subroutine is a subroutine call of the following form:

subr(arg_1, arg_2, arg_3, ...),

where subr is the subroutine name and arg_i is a symbol for the ith argument to the function. Arguments are specified as either subroutine inputs or outputs. All subroutine symbols are linked to quantity attributes of domain objects, exactly as is done with equation symbols. However, as described in Section 3.2.4, the execution of a subroutine is handled by a foreign function call, rather than by SIGMA's model-interpreter. Some additional information is needed to interface SIGMA properly with external subroutines. In particular, it is necessary to specify the dimensionality of any array structures expected by the subroutine as input or produced as output. This is not necessary for equations, because the model-interpreter can handle scalar or vector structures as required. With subroutines, it is also necessary to specify the scientific units expected by the input and output arguments. When executing equations, SIGMA handles scientific unit conversion transparently by converting all inputs to a common scientific unit scheme before calculating the result. This is not possible with externally-written code that rigidly expects input values to be in terms of specific scientific units and produces output values that carry no scientific units. A detailed example of SIGMA's subroutine representation can be found in Appendix Section A1.5.

SIGMA's representation for equations and subroutines can be viewed as defining a small constraint network that provides the semantics for the symbols in the transform. To determine whether a

transform applies in a given situation, the network must be matched to the situation and the constraints must be checked. We will describe this matching process in greater detail when we revisit the SIGMA guided tour in Section 6. This matching process depends heavily on the subsumption relationships encoded in SIGMA's scientific knowledge base, which is the topic of the next section.

5 SIGMA's domain knowledge

SIGMA's knowledge base provides the necessary background knowledge to support the system's model construction activities. This background knowledge is represented and stored in a hierarchically organized, frame-structured knowledge base. The objects in this knowledge base encode a variety of different types of knowledge, including information about scientific equations, physical quantities, physical constants, scientific units, scientific domain concepts, and bibliographic citations. The knowledge base contains information pertinent to scientific computing in general, as well as information specific to our two main application areas: planetary atmospheric modeling and forest ecosystem modeling. At present, the knowledge base contains over 2000 objects. These objects contain definitions for approximately 600 classes which are described in terms of over 400 attributes. There are more than 1000 instances[5] of these 600 classes in the initial background knowledge base. Additional problem-specific instances are added to the background knowledge when an experiment scenario is instantiated.

Overall, SIGMA's knowledge can be partitioned into four categories:

1. **Cross-disciplinary scientific knowledge:** Knowledge available to persons with a general scientific background, including knowledge about various physical quantities, scientific domain objects, scientific measure units, fundamental equations, and scientific handbook data.
2. **Discipline-specific scientific knowledge:** Quantities, domain objects, equations, and data pertaining to a specific scientific discipline (e.g., biology, ecology, physics).
3. **Problem-specific knowledge:** Domain objects and relations pertaining to the specific physical system being modeled by the scientist.
4. **Programming knowledge:** Knowledge about numerical programming methods, data structures, control, etc. (In the current version of SIGMA, much of this knowledge is implicit in the model-interpreter.)

A partial overview of the knowledge base is depicted in Figure 17. Although this overview is incomplete, it gives a feeling for the structure of SIGMA's knowledge base. The construction of this knowledge base represents a considerable expenditure of resources on our part. Our attention to

[5]The large majority of these instances represent physical constants associated with chemical molecules and solar system objects defined in the knowledge base.

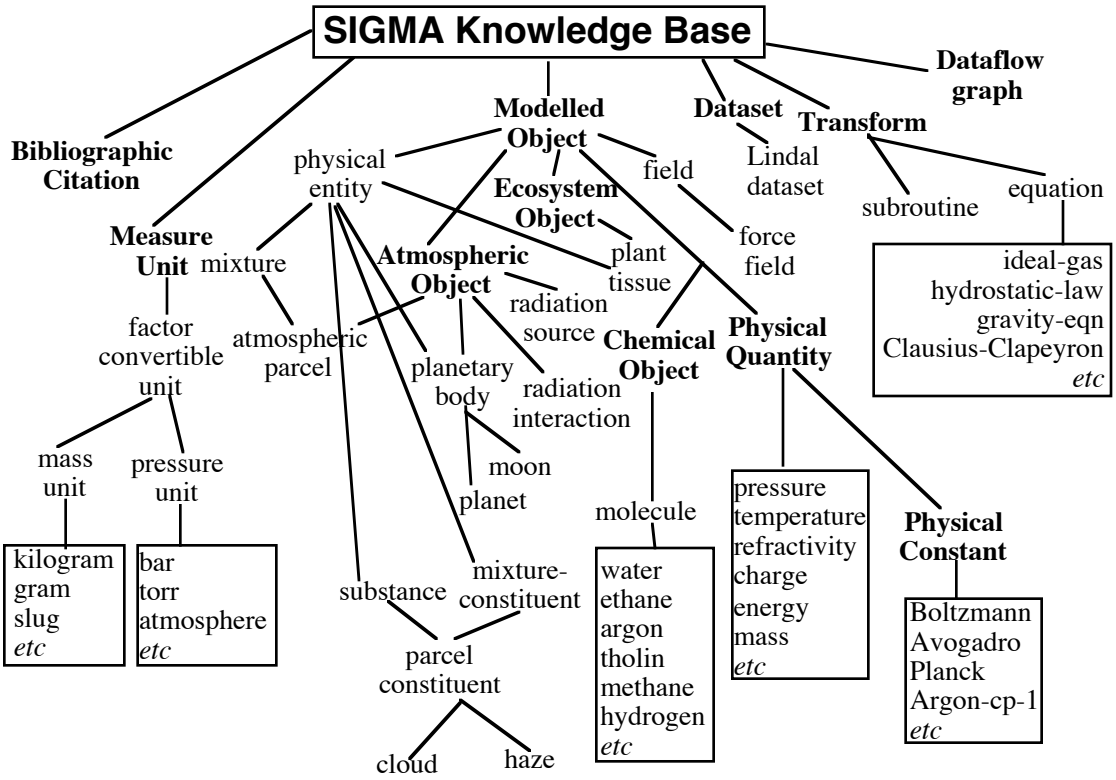

Figure 17: Overview of SIGMA's knowledge base

construction of the knowledge base and the associated ontology is motivated by the goal of reuse; We want to make the knowledge reusable across a variety of scientific modeling domains. For example, the concept of a physical-entity is general enough that it can be used in almost any modeling domain. Physical-entity provides the basic types of quantities you would associate with any physical thing: mass, density, temperature, etc. Domain objects such as 'atmospheric-parcel' in the atmospheric sciences domain and 'plant-tissue' in the ecosystem sciences domain inherit from this general concept.

The hierarchy depicted in Figure 17 illustrates the subsumption relationships in SIGMA's knowledge base, but does not give a feeling for the highly interconnected nature of the objects in the knowledge base. As mentioned before, each scientific domain object in the knowledge base has two types of slots: quantity slots and link slots. The quantity slots store physical quantity instances associated with the object; the link slots point to other related domain objects. Each physical quantity instance in turn contains two basic slots: a value slot and a units slot. The value slot stores the quantity's magnitude; the units slot stores a unit expression associated with the value. The unit expression must be consistent with the defined unit type associated with the definition of the quantity. For example, instances of the quantity 'pressure-quantity' must have an associated instance of type 'pressure-unit'. Instances of pressure-unit can be either compound units of the form *force-unit/area-unit* (e.g., *dyne/square-meter*) or

primitive pressure units (e.g., *bar*). SIGMA's knowledge base contains information that permits it to convert units as necessary for consistent calculation. For example, the object representing the primitive pressure unit 'bar', contains conversion factors necessary to convert bar to pascal, atmosphere, or to any other known primitive pressure-unit.

Appendix Section A1.1 provides sample object definitions intended to give the reader more insight into the structure of the objects in SIGMA's knowledge base. Due to its size and complexity, a complete description of the knowledge base structure is beyond the scope of this paper.

SIGMA's representation language, called RML (Nayak, 1992), is built on top of CommonLisp and CLOS, the CommonLisp object system. RML includes various inference features, including inheritance and a forward-chaining rule system. RML also contains an integrated constraint language that enables specification and efficient checking of arbitrary first order constraints involving slot values and instances in the knowledge base. The RML representation language is based on CYCL, the language used in the CYC project (Guha & Lenat, 1990).

6 The guided tour revisited

Having described SIGMA's equation representation language and the structure and contents of SIGMA's knowledge base, we are now in a position to revisit the SIGMA guided tour presented in Section 3. Our focus this time will be on understanding how SIGMA's underlying representation and inference mechanisms work to ensure scientific coherence during model-building. Recall from Section 3.3 that a model is considered scientifically coherent if all transforms in the model are appropriately applied and if the transforms interconnect to form a globally consistent computation.

Operationally, SIGMA enforces scientific coherence by limiting the set of transforms that can be applied at any point to extend a model. Recall that model-building is accomplished by iteratively expanding a data flow diagram to connect unknown quantities with known quantities via a sequence of equations or subroutines. The user clicks the right arrow button to choose the transform that should be applied to compute a desired unknown quantity. The set of candidate transforms presented to the user is determined by starting with all of the transforms defined in SIGMA's knowledge base, and then filtering out those that are inappropriate to apply in the current modeling situation. This transform-filtering procedure is at the heart of SIGMA's intelligent functionality.

6.1 Transform-filtering procedure

SIGMA's transform-filtering procedure ensures that the user will only be permitted to augment the current model by applying an equation or subroutine that preserves scientific coherence. The set of eligible transforms is computed as follows. Starting with the set of all transforms T in the knowledge base, SIGMA first computes the subset of syntactically applicable transforms (T_{syntax}). To determine this set, each transform is accessed to determine whether any symbol in the transform has the same quantity type as the desired unknown quantity, and if so, whether that symbol can be calculated as the output of the transform. Transforms that pass this test are included in the subset T_{syntax} . From this restricted subset, SIGMA next determines the subset of semantically applicable transforms ($T_{semantic}$). These include only transforms from T_{syntax} that also match the semantic constraints imposed by the transform representation. Finally, $T_{semantic}$ is reduced further to create $T_{consistent}$. This set eliminates transforms from $T_{semantic}$ that are inconsistent with choices made in the model fragment constructed by the user thus far. To summarize, $T \supseteq T_{syntax} \supseteq T_{semantic} \supseteq T_{consistent}$. The user's choices are restricted to $T_{consistent}$.[6] Each phase of the transform-filtering process is covered in more detail in the next section.

6.2 Transform-filtering in practice: the SIGMA guided tour

Consider an illustration of how the transform-filtering procedure works in practice. This example is based on the interaction described in Section 3.2.3 of the SIGMA guided tour. In this interaction, the user wishes to compute the density of the set of Titan atmospheric parcels.

To compute T_{syntax}, the system searches to find all transforms containing a symbol with the same quantity type as the density goal quantity. Because each symbol in a transform is linked to a quantity slot of some object, and because these quantity slots are typed according to physical quantity, this search process is simple. Consider the Gravitational Force Equation in Figure 16. This equation is not included in T_{syntax} because none of its symbols is linked to an attribute with the same type as the density quantity. But even if a correctly-typed symbol is found within a given transform, this does not guarantee that the transform will be included in T_{syntax}. The symbol must also be computable as an output of the transform. This condition may not be satisfied in one of two cases. First, the equation may not be mathematically solvable for the symbol. SIGMA uses the REDUCE (Hearn, 1991) computer algebra system to symbolically manipulate each equation formula to isolate the target symbol on one

[6] With respect to SIGMA's user interface, note that the user sees only two sets of transforms in the transform selection window (Figure 6): "Applicable Transforms" and "Near Miss Transforms". The applicable transforms correspond to those in $T_{consistent}$. Thus, the user is restricted to choices that are guaranteed to yield a scientifically coherent model fragment. The "near miss" transforms correspond to inconsistent equations, i.e. those in $(T_{syntactic} - T_{consistent})$.

side. If the symbol cannot be isolated, the equation is rejected from T_{syntax}. Second, it is possible that the matching symbol in the transform is not designated as a "possible-output" symbol. Typically, all symbols in an equation are designated as possible outputs unless explicitly excluded by the domain scientist who entered the equation.[7] For subroutines, only certain arguments are designated as outputs.

One example of a transform that is included in T_{syntax} is the equation entitled "Refractivity of a Gaseous Mixture":

$$\mathbf{R = \Sigma_i (N_i * p_i)}. \quad (2)$$

The graphic representation for this equation is shown in Figure 18, and the internal representation is given in Appendix Section A1.3. This equation is included in T_{syntax} because it can be expanded and algebraically manipulated to compute a density quantity (N_i for some i).

After T_{syntax} is computed, the next step in the transform-filtering procedure is to check whether each transform in this set satisfies its defined semantic constraints in the context of the current experiment scenario. For example, it turns out that applying Equation 2 in the context of the Lindal experiment scenario is semantically justifiable. To see why, it is first necessary to understand the scientific meaning of the equation, as depicted in Figure 18. This equation computes the amount of refractivity induced when a mixture and its constituents are irradiated by an electromagnetic radiation source. In Figure 18, there is a separate radiation-interaction object to represent the collision of the source with the constituents and with the entire mixture. The refractivity (R) is computed by taking a sum over each constituent (i) in the mixture, and calculating the product of the constituent's density (N_i) and its induced polarizability (p_i).

To determine whether the semantic constraints associated with Equation 2 are satisfied, SIGMA matches the general constraint network in Figure 18 to the instantiated objects in the Lindal experiment scenario (Figure 3). This matching process utilizes the class-subclass subsumption relationships specified in SIGMA's knowledge base (Figure 17). In the case of Equation 2, the matching process finds the following legitimate bindings between objects in the equation's constraint network and instances in the experiment scenario: [8]

---

[7] Suppose an equation represents a new term definition in which a second-order quantity is defined in terms of more fundamental quantities. The fundamental quantities are measurable, but the second-order quantity is only calculable in terms of the fundamentals. Such equations are often written for notational convenience. Although solving the equation for one of the fundamental quantities may be mathematically possible, it is of questionable operational value to calculate a fundamental quantity from a non-measurable second-order quantity.

[8] Because the user specified that the scope of the computation covers the entire grid array of atmospheric parcel objects (see Section 3.2.2), SIGMA actually binds the objects in Equation 2 to arrays of instances, rather than to the single instances identified from Figure 3. For ease of understanding, we describe the matching process only in terms of the representative subset of instances displayed in Figure 3.

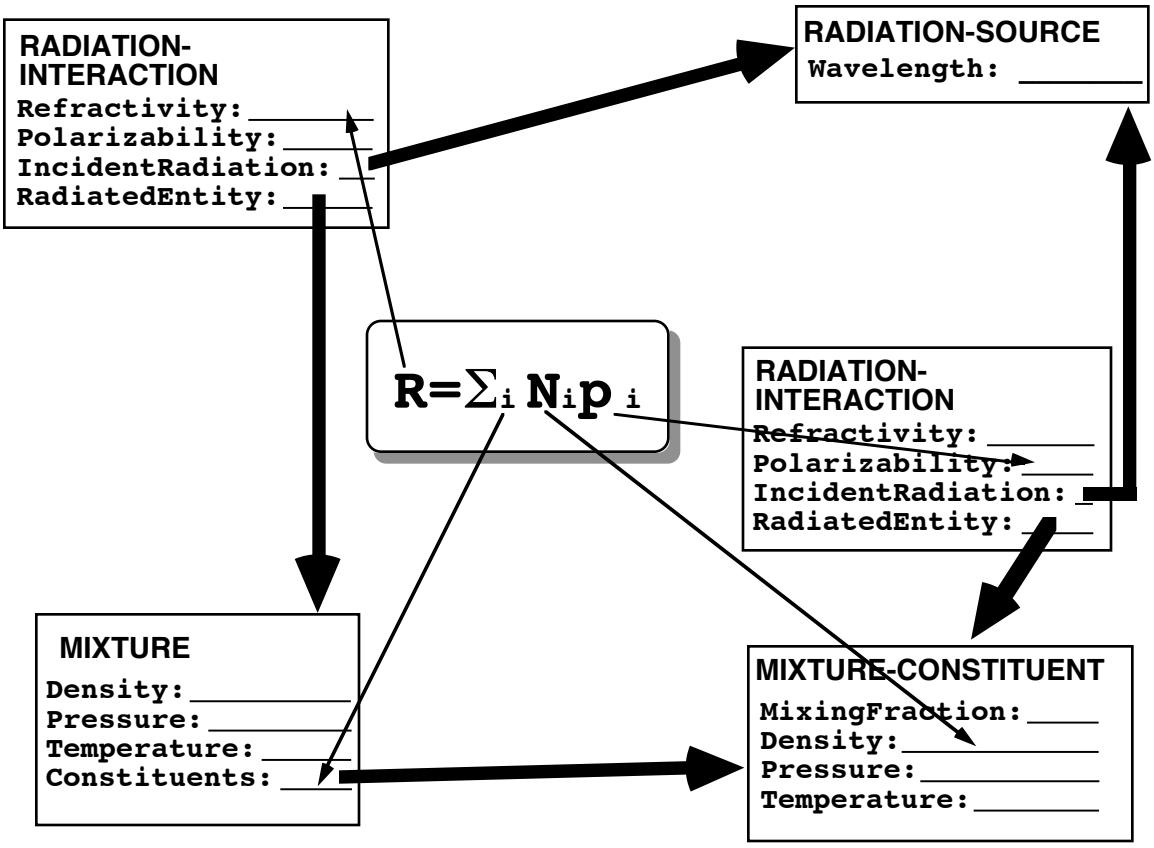

Figure 18: Representation for equation entitled "Refractivity of a Gaseous Mixture"

Mixture = Atmospheric-Parcel-32;

Mixture-Constituent = H2-Constituent-32 and N2-Constituent-32[9];

Radiation-Source=Voyager-Signal;

Radiation-Interaction(w/Mixture) = Radiation-Interaction-14;

Radiation-Interaction(w/Constituent) = Radiation-Interaction-15 and Radiation-Interaction16.

Because there is a valid match, the semantic constraints are satisfied and Equation 2 is included in $T_{semantic}$.

The final test for equation applicability is to check that equation application would be consistent with the choices made in the model fragment constructed thus far. This is where Equation 2 runs into trouble. The user has selected the density of the set of atmospheric parcels as the goal quantity to be computed. According to the subsumption hierarchy in Figure 17, an atmospheric-parcel is a type of

[9]Note that there are two matching instances for both Mixture-Constituent and Radiation-Interaction (w/Constituent). This is due to the summation construct in Equation 2, which sums over each of the constituents. At the implementation level, SIGMA explicitly expands the summation construct and creates distinct object bindings for each constituent.

mixture. However, based on the semantic constraints expressed in Figure 18, the symbol N_i in Equation 2 corresponds to the density of a mixture-constituent, not to the density of a mixture. In fact, no symbol in the equation corresponds to the density of a mixture. So although Equation 2 is appropriate for computing the density of a constituent within some mixture (e.g., the density of the hydrogen gas within the atmospheric parcel), it is inappropriate for computing the density of the mixture itself (e.g., the density of the entire atmospheric parcel). Thus, Equation 2 must be omitted from $T_{consistent}$.

Now consider the "Density Computation" transform which was actually selected by the user to compute the density of the atmospheric parcel in the guided tour:

$$\mathbf{N = R / p}. \tag{3}$$

This equation describes the density of a physical entity as the quotient of the refractivity and the polarizability induced when the entity is irradiated by an electromagnetic radiation source (see Figure 19 and Appendix Section A1.4 for the corresponding representation). In defining this equation, the distinction between a mixture and a constituent is irrelevant. Equation 3 applies to any physical entity, regardless of whether the entity is a mixture or part of a mixture. In particular, Equation 3 applies to computing the density of atmospheric parcels. The matching process yields the following sets of bindings between the objects in Equation 3 and the experiment scenario instances:

Set #1:
Physical-Entity = Atmospheric-Parcel-32;
Radiation-Source=Voyager-Signal;
Radiation-Interaction = Radiation-Interaction-14;

Set #2:
Physical-Entity = H2-Constituent-32;
Radiation-Source=Voyager-Signal;
Radiation-Interaction = Radiation-Interaction-15;

Set #3:
Physical-Entity = N2-Constituent-32;
Radiation-Source=Voyager-Signal;
Radiation-Interaction = Radiation-Interaction-16;

Sets 2 and 3 are rejected for the same reason that Equation 2 was eliminated above: in these cases, the symbol N in Equation 3 references the density of the wrong object in the experiment scenario. But in Set 1, N does match the desired goal quantity -- the density of the atmospheric parcel. Thus Equation 3 is included in $T_{consistent}$.

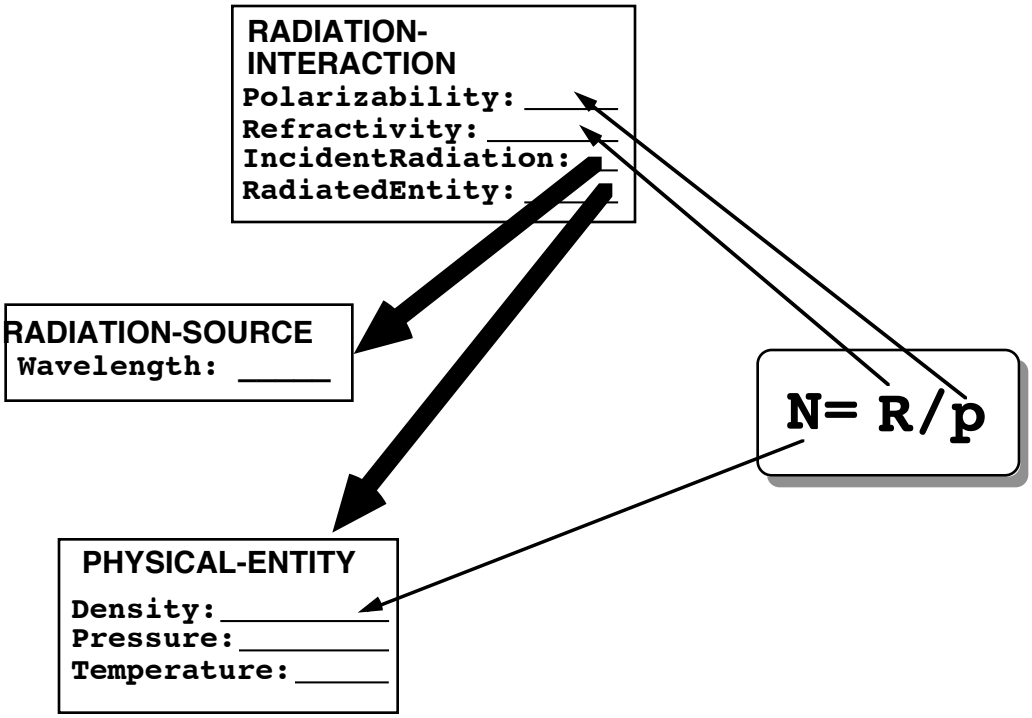

Figure 19: Equation representation for Density Computation

6.3 Maintaining global consistency

Having gained some insight into how individual equations are matched to ensure scientific coherence, we continue with our reexamination of the SIGMA guided tour. At this point, our aim is to understand more about how the transform-filtering procedure enforces scientific coherence globally over the entire model. Essentially, this is done by propagating constraints throughout the model to ensure consistency across all transforms. Because the output from one transform in a model is the input to another, the semantic constraint networks for the individual transforms in the model can be logically unified through their inputs and outputs. When taken as a whole, these individual transform-level constraint nets can be composed to form one large constraint network underlying the entire model. We illustrate with the next step in the guided tour.

The second equation selected by the user in the guided tour is the equation entitled "Polarizability of a Gaseous Mixture" (see Figure 20):

$$\mathbf{pol} = \Sigma_{\mathbf{i}}(\mathbf{f_i} * \mathbf{p_i}). \quad (4)$$

This equation computes the polarizability induced when electromagnetic radiation intercepts a mixture. The polarizability (pol) associated with the entire mixture is computed from the

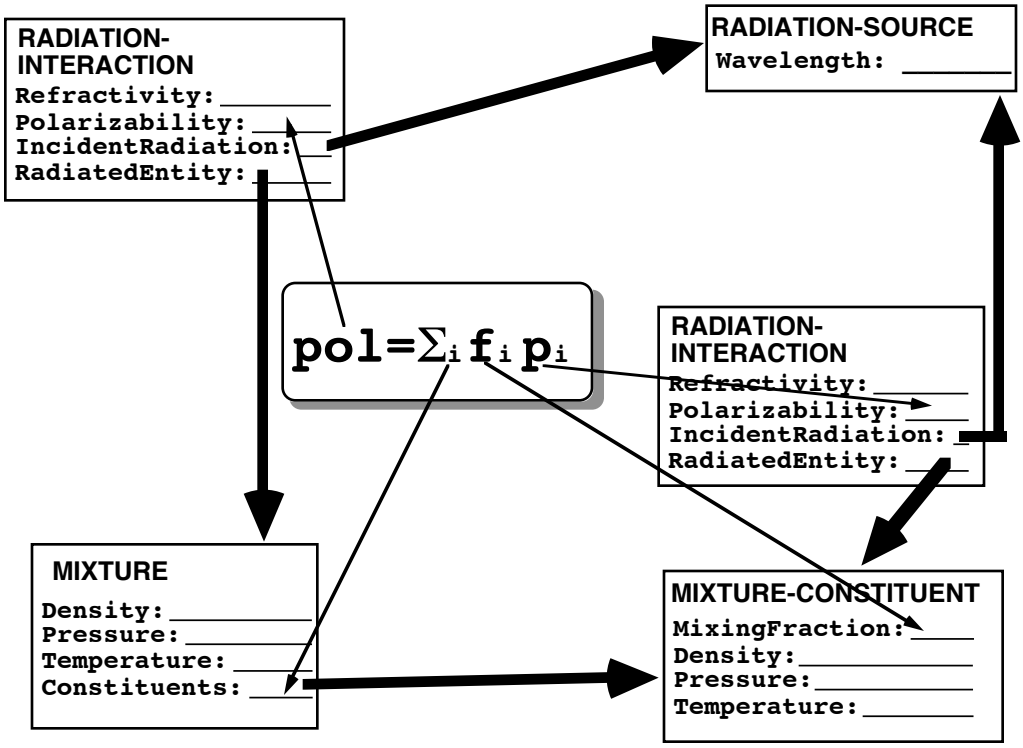

Figure 20: Representation for equation entitled "Polarizability of a Gaseous Mixture"

polarizability associated with each constituent (p_i), weighted by its fraction (f_i) within the mixture. Equation 4 is selected by the user to compute the polarizability of the atmospheric parcel. This output of Equation 4 is required as an input to Equation 3. The input/output correspondence provides the conduit through which constraints from Equation 3 flow to Equation 4.

In matching Equation 4 to the experiment scenario, there is initially some ambiguity concerning which equation symbol should be treated as the output. Even though there are two symbols representing polarizability quantities in Equation 4 (pol and p_i), only one of these symbols makes semantic sense as the output in this situation. In particular, pol is the correct output because it references the polarizability of a mixture, and an atmospheric parcel is a kind of mixture. On the other hand, P_i is an inappropriate output candidate because p_i references the polarizability of a mixture-constituent, and the atmospheric parcel is not a kind of constituent.

Once the correct output symbol is determined by the system, the matching procedure logically unifies the (input) symbol p in Equation 3 with the (output) symbol pol in Equation 4 to ensure that they both reference the same quantity. In particular, the mixture referenced by Equation 4 must be the same as the physical entity referenced by Equation 3. This is enforced by propagating the binding established for the physical-entity to the mixture. In binding set #1 for Equation 3, the binding for physical-entity was

established as atmospheric-parcel-32. This binding is propagated to the mixture. Once the mixture binding is established, this constrains the other objects in Equation 4 so they are consistent with the choice propagated from Equation 3. SIGMA maintains this type of global consistency in interpretation throughout the entire model.

In some cases, there may be more than one way in which a single transform might be consistently applied to the experiment scenario. This happens when the matching process produces more than one binding set (as with Equation 3 above). If there are an insufficient number of constraints propagated to narrow the selection to a single consistent binding choice, SIGMA queries the user to select the appropriate interpretation. For example, suppose the user wishes to compute the density of some constituent within an atmospheric parcel using Equation 3. Because there are multiple constituents, the system may not be able to determine which constituent the user intends to reference without further guidance.

6.4 Discussion

Although the representation and inference mechanisms for maintaining scientific coherence are somewhat complex, most of this complexity is transparent to the user. From the user's point of view, the process of model-building is accomplished by graphically constructing a simple data flow diagram. The user is unaware that the system simultaneously constructs a complex underlying constraint network to assist in maintaining scientific coherence. The constraint propagation and maintenance mechanisms that are employed by the system have a significant beneficial side-effect: they reduce the number of choices that the user needs to make. For example, without the availability of constraints, the system would propose applying transforms in situations where they are semantically meaningless to apply, and the user would be left to sift through a set of unrealistic choices when only one choice is actually feasible. This would increase user frustration and decrease the overall utility of the tool.

In addition to the constraint mechanisms, SIGMA's transform representation decreases the amount of user interaction necessary with the system, as well. Aside from defining when transforms apply, SIGMA's transform representation also provides the information necessary to properly apply the transforms and bind the symbols in the formula. With the availability of this information, SIGMA relieves the user from the burden of exhaustively specifying how each symbol in a modeling transform binds to a quantity in the experiment scenario. In effect, the transform constraints enable the system to infer part of the model specification automatically. Consider the application of the Density Computation equation, as discussed in Section 6.2. To unambiguously specify how to apply the equation, the user must only identify the atmospheric-parcel instance to which the equation should be applied. The system uses its representation of the generic equation (Figure 19) to determine which subsidiary

instances are involved in equation application. Subsidiary instances are identified by following links from atmospheric parcel instance to all other instances referenced by the generic equation. Note that a SIGMA user never explicitly needs to specify bindings for any of the symbols in the equation; based on the semantic annotations, the system can automatically infer which quantity in the experiment scenario the user intends to bind to a given symbol. And once initial bindings are established, they are automatically propagated through the data flow diagram, thus relieving the user of the tedium of repeatedly specifying bindings when they can be easily inferred based on the structure of the diagram and the semantic annotations.

In summary, SIGMA's transform representation and constraint propagation mechanisms are the keys to the system's ability to provide intelligent support in model-building. They work to increase the system's autonomy and reduce the user's need to unnecessarily interact with the system.

7 Related work

On the surface, SIGMA appears similar to a large class of data flow based visual programming environments that have been developed recently. There is a significant body of research on visual programming environments and methods of representing program structure and content graphically (Glinert, 1990). The recently-developed tools help users graphically construct software in a variety of application areas, including image processing and scientific visualization (apE, 1992; Atwood, Blankenbecler, Kunz, Mours, & Weir, 1990; AVS, 1992; Iconicode/IDF, 1992; Khoros, 1992), scientific instrument design (LabVIEW, 1992), and simulation (Extend, 1992; STELLA/IThink, 1992). In all of these cases, however, the software tool has fairly limited knowledge of the application domain. Although the tools enforce simple syntactic checks on the data flow graphs and perform some type-checking, none of these tools has a deep semantic understanding of what the data flow program is doing and whether the operations on the data make sense. As a result, it is possible with these tools to create a syntactically valid flow graph that is semantically meaningless to a domain specialist. Other scientific computing tools, including symbolic algebra systems (e.g., Mathematica (Wolfram, 1988) and Maple (Char, 1988)) and data analysis tools (e.g., MATLAB (MATLAB, 1989), IDL (IDL, 1992)), suffer from similar problems related to their lack of knowledge about the scientific problem to which they are being applied. In contrast, SIGMA uses domain knowledge during the model-building process to check the model for scientific coherence as it is being constructed. In this way, SIGMA ensures that the user constructs a semantically meaningful program.

For purposes of comparison with SIGMA, ThingLab (Borning, 1981) is perhaps a more relevant visual programming environment to consider because it employs a knowledge-based approach. ThingLab is a "simulation laboratory" that shares with SIGMA the idea of representing and visualizing constraints

on domain objects in a simulation model. ThingLab operates on a variety of different types of constraints, including symbolic as well as numeric constraints, and uses general constraint satisfaction methods to calculate unknown quantities from known quantities in its constraint networks. Many of SIGMA's constraint mechanisms have a precursor in the ThingLab system.

Aside from work on visualization, our work on SIGMA bears a close relationship to the growing body of work on applying knowledge-based techniques to various aspects of scientific and engineering computation (Abelson, Eisenberg, Halfant, Katzenelson, Sacks, Sussman, et al., 1989; Barstow, 1985; Cook, 1990; Ford & Chatelin, 1987; Kook & Novak, 1991; Palmer & Cremer, 1991; Robertson, Bundy, Uschold, & Muetzelfeldt, 1989; Weld & deKleer, 1990). We review some of the particularly relevant systems for comparison below.

The ECO project (Robertson, Bundy, Muetzelfeldt, Haggith, & Uschold, 1991; Robertson, et al., 1989) shares many goals with SIGMA. The ECO program was designed to enable ecologists to build simulation models using a special-purpose sorted logic designed to express ecosystem concepts. The models are conveyed in terms of the Systems Dynamics formalism (Forrester, 1961), which uses "reservoirs" and "flows" to express differential equations. SIGMA and ECO differ primarily in terms of their representational power and their scope. SIGMA uses a general-purpose frame-based representation to express domain concepts, whereas ECO uses a more expressive, specialized logic formalism that we believe would be more difficult for scientists to use. SIGMA has been designed as a general model-building assistant, intended to assist scientist in a number of domains, whereas ECO is more narrowly scoped to ecological models and the Systems Dynamics formalism.

Another system that is very close to SIGMA in both its goals and approach is SIMLAB (Palmer & Cremer, 1991). Like SIGMA, SIMLAB provides users with a high-level language for specifying physical system models. This language allows users to specify objects, quantities, and constraints in the modeling scenario, just as in SIGMA. SIMLAB can represent more numerically sophisticated models than SIGMA, however. SIMLAB can handle simultaneous sets of arbitrary ordinary differential equations, whereas SIGMA only handles a non-coupled first-order ODEs. SIMLAB's model specifications are translated into Fortran code that invokes differential equation solver packages necessary to solve the set of equations. In contrast, SIGMA's model structures are interpreted directly within LISP. The two systems also differ in how equations are applied to objects in the experiment scenario. SIGMA users specify equation application by interactively constructing a visual data flow diagram, whereas SIMLAB users must write pseudo-code to apply equations within the experiment scenario. Perhaps the key feature that distinguishes SIGMA from SIMLAB is the use of a knowledge base to represent and store scientific domain knowledge as part of a permanent, reusable repository. The SIGMA user does not need to specify details associated with quantities and units because this information is available in SIGMA's knowledge base. Also, SIGMA is designed as an interactive

assistant that dynamically helps the scientist with the process of model-building. SIMLAB simply provides a high-level language and does not assist in the model-building process. Finally, SIGMA's explicit representation of semantic equation annotations appears to be more explicit and powerful than SIMLAB's representational scheme.

ϕ_0(Barstow, Duffey, Smoliar, & Vestal, 1982) is a domain-specific automatic programming system constructed at Schlumberger to assist in generating oil well log interpretation software. The system was designed for direct use by petroleum scientists, who were to use it to construct geological models expressed as a set of quantitative equations relating geological parameters of interest. Like SIGMA and ECO, ϕ_0 makes extensive use of scientific domain knowledge to aid in the model-building process. However, SIGMA's equation representation appears to incorporate more domain knowledge and constraints than the representation used in the ϕ_0 prototype.

In a similar vein, (Kant, Daube, MacGregor, & Wald, 1991) describes the SINAPSE system under development at Schlumberger. SINAPSE helps scientists build mathematical modeling software used in the context of data interpretation tasks such as seismic interpretation. In particular, the system synthesizes finite-difference programs that implement partial differential equation models. SINAPSE can be compared with SIGMA on a variety of dimensions. In SIGMA, the emphasis is on specification acquisition, rather than code synthesis; in SINAPSE, the emphasis is reversed. The grainsize and nature of the scientific modeling problem addressed by the two systems is quite different. SIGMA provides assistance in specifying a complete model from end to end, whereas SINAPSE focuses on the subtask synthesizing efficient algorithms for modeling steps involving the solution of partial differential equations. Conceptually, SINAPSE could be called as a subroutine to synthesize code for one of the modeling steps specified using SIGMA. Finally, the "domain" of expertise for these two systems is different. SIGMA's "domain knowledge" consists of knowledge about the quantities, equations, objects, and constraints necessary to model a particular physical system of interest to the scientist. SINAPSE's "domain knowledge" consists primarily of knowledge about mathematics, rather than knowledge about the scientific domain under study.

Our motivation and approach is similar in spirit to the philosophy behind ARIES (Johnson & Harris, 1991). ARIES helps requirements analysts construct specifications by providing defined concepts and terminology which are stored in a large knowledge base that employs representational mechanisms similar to those available in SIGMA's representation language. The ARIES knowledge base stores knowledge about the air traffic control domain. SIGMA can be viewed as an additional instance of a new class of software tool that uses extensive domain knowledge to assist with various aspects of the software engineering task (Barstow, 1985; Iscoe, 1991).

Our work also bears some relationship to a growing body of work on automated model construction (Addanki, Cremonini, & Penberthy, 1991; Falkenhainer & Forbus, 1991; Ling & Steinberg, 1992; Nayak,

1992; Weld, 1990). This work originates from within the model-based reasoning community, where the focus has been primarily on qualitative modeling of physical systems. Several of these systems are based on the notion of compositional modeling (Falkenhainer & Forbus, 1991), in which model fragments are assembled together to build whole models. Model fragments encapsulate a set of relevant equations that describe some physical phenomenon. Although the concept of model fragments may prove useful in SIGMA, the annotated data flow diagram representation currently emphasizes individual equations rather than aggregated fragments as the central organizing structure behind modeling. One significant difference with respect to the automated modeling work is that SIGMA provides only model-building assistance, not complete automation. The user must make choices as to which equations should be included in a given model. We believe that the size of the search space for building models of reasonable complexity will preclude a completely automated approach. Model-building is inherently an open-ended activity, and most automated approaches rely heavily on closed-world assumptions in which the equations, objects, and relations are all known in advance of the model formulation activity. In areas of active scientific investigation, this assumption may be ill-advised and inappropriate. In any case, we are focusing on providing intelligent assistance to the scientist, and may consider additional steps toward automation as we become convinced of the feasibility and user acceptance of this approach.

8 Status and future work

We have successfully used SIGMA to reimplement portions of two pre-existing Fortran models -- one from an ecosystem sciences domain and one from a planetary atmospheric sciences domain. In both of these cases, the models created with SIGMA have replicated results previously published in the scientific literature (Lindal, et al., 1983; Running & Coughlan, 1988). To give a feeling for the size of these models, the ecosystem model consists of about 50 transforms that calculate or use over 60 different quantities, while the Titan atmospheric model consists of about 25 transforms using or calculating 45 quantities. (These 45 quantities are replicated over a gridded array of dimension 100.) Both of these models execute in real time, although the entire Titan model takes on the order of tens of seconds elapsed time to execute. In this case, the inefficiency is due to non-optimized code and an untuned runtime LISP environment; The portion of the Titan model we have implemented is not intrinsically computationally complex.

SIGMA is implemented in CommonLisp on a Sun SPARCstation 2, and features a direct manipulation Motif-based graphical user interface. The interface was constructed with a user interface-building environment called GINA (Spenke & Beilken, 1990). SIGMA's development to date represents over four person-years of effort.

SIGMA is currently undergoing a series of tests by our science collaborators to evaluate its promise as a working tool. We feel the long-term prospects for this work are very encouraging. Scientists' initial exposure to working with SIGMA -- although limited -- has been extremely enthusiastic. The scientists seem to appreciate the benefits of imposing a rigorous structure on what is typically an ill-defined and ad hoc model-building process. Among the perceived benefits are automated assistance and more sophisticated error-checking capabilities. Aside from its primary role as a model-building tool, users have expressed great interest in SIGMA as a training tool and as a sophisticated documentation tool for scientific models.

Our use of data flow diagrams as a language for describing models has been particularly well-received by users. In several instances, scientists were surprised and enlightened upon viewing their old Fortran modeling code expressed in this type of abstract, graphical representation. There are several reasons for this enthusiasm. First, data dependencies are much easier to trace in graphical form than in textual form. Data dependencies are a critical to solid understanding of model behavior. Users also have been enthusiastic about the direct and immediate accessibility of their code using the SIGMA interface. The interface allows them easy access to intermediary computed values that appear as quantity nodes in the data flow diagram. This easy access makes debugging a simple matter of selectively probing quantity nodes in the data flow diagram, and tracing the dependency links backwards toward known values. In much the same way as schematic diagrams are a convenient notational language for specifying electromechanical devices, data flow diagrams serve a convenient, structured medium for model-building. The diagram structure allows users to construct models in a piecemeal fashion and avoids the type of linear thinking associated with conventional programming.

Based on our experience to date, it appears that there are several issues that need to be addressed before the system can be used in a routine fashion for modeling. We discuss these issues in the following sections.

8.1 Efficiency, numerical sensitivities

As we have mentioned, the current version of SIGMA focuses only on prototyping scientific modeling software, and ignores issues related to efficient code synthesis, numerical stability, database access, and platform dependencies. For applications where such concerns are minimized, models can be built with SIGMA in a fraction of the time necessary to produce the corresponding Fortran code, and executed to produce useful results. But in many cases, it will be necessary to compile more efficient and more numerically-sophisticated modeling code. We envision using SIGMA's internal model representation as the starting point for a more complete, end-to-end code generation process.

8.2 Knowledge acquisition and maintenance

Another concern is the lack of adequate knowledge acquisition tools to facilitate knowledge entry by the scientist-user. We must design and build acquisition and browsing tools to help users inspect, augment, and maintain the knowledge in SIGMA' s knowledge base. In realistic modeling interactions, the user will need to add new quantities, scientific units, objects, attributes, and equations. Without the ability to add new knowledge, modelers quickly run into dead ends. It is unrealistic to expect anyone to anticipate the needs of the modeler and build a 'complete' knowledge base.

We have recently taken an important step toward addressing knowledge acquisition concerns by implementing a graphical equation editor that permits users to enter and modify equations and their associated semantic augmentations. Additional problems have surfaced with the introduction of this editor. In particular, the user must understand some aspects of SIGMA's knowledge representation scheme in order to construct an equation annotation. For example, the user must attach each symbol in a new equation to an attribute of some object in SIGMA's knowledge base. Although well-motivated by the goals of representational soundness and knowledge reuse, the structure of the knowledge base is complex and at times unintuitive to the domain specialist. The knowledge base structure may prove to be a barrier to successful knowledge acquisition unless we provide methods for making these representation structures intuitive and easy to understand.

8.3 Model complexity

Once users have the ability to augment SIGMA' s knowledge base, the size and complexity of models that can be constructed with the system will grow quickly. As the number of quantities and transforms in a model grows, it becomes increasingly difficult to visualize and manipulate the model graphically. We will need to provide tools and techniques to manage this complexity. We have explored several methods of displaying and browsing the large data flow diagrams that users construct with SIGMA.

One promising technique we intend to pursue is allowing users to hierarchically abstract portions of the data flow diagram. In effect, this will allow users to package meaningful computational units (e.g., sequences of transforms) within a model. We also plan to allow users to export these packaged computations as types of reusable macro-transforms within SIGMA's knowledge base.

8.4 Experiment scenario configuration

SIGMA does not currently provide support for configuring experiment scenarios. Instead, SIGMA contains a set of pre-defined experiment scenarios developed in collaboration with domain scientists and knowledge engineers. These scenarios may be parameterized, so each scenario generates a

potentially large number of instantiated scenarios. However, each pre-defined scenario configuration is fixed, and can only be changed by system designers. The initial idea behind fixing the scenarios was that each scenario would capture a particular scientist's modeling expertise, and portray his or her individual view of the experimental situation. Since users of the system would generally have less expertise than the expert, we reasoned that they would willingly adopt one of these expert viewpoints. In practice, our collaborators are experts, rather than novice model users, and would like the ability to configure the experiment scenario.

8.5 Multiple models

Modeling is inherently an iterative, exploratory process. A scientist may formulate one or more hypotheses, construct models to test these hypotheses, compare results across models, modify hypotheses, and begin again. SIGMA provides a mechanism for scientists to implement one model at a time, but does not allow them to construct multiple models simultaneously or to compare results across models. We have begun to implement a kind of "multiple worlds" mechanism (Filman, 1988; Guha, 1991) that will permit users to have this type of capability.

8.6 Mathematical sophistication

As mentioned previously, SIGMA limits the type of mathematics that can be used in a model. In particular, SIGMA supports only non-coupled algebraic and first order ordinary differential equations. This restriction severely limits the mathematical sophistication of the models that can be specified using SIGMA. We would like to extend SIGMA's capabilities to cover simultaneous equations, including higher-order ordinary and partial differential equations. To handle these cases, we will need to address issues of discretization and grid geometries that are not addressed in the current system.

8.7 Model validation

Validation of results is an important part of the model-building process. Model results must be validated against known experimental data and *a priori* expectations, including numerical trend expectations, sign and order of magnitude expectations, and domain-specific expectations. Sensitivity analyses must be conducted to determine the robustness of the model in the face of small changes to inputs. SIGMA provides some incidental support for the validation process (e.g., the ability to change input parameters and re-execute a model), but this support must be improved and conveniently packaged.

9 Summary and conclusions

In this paper, we have described SIGMA, a knowledge-based software development environment for prototyping scientific models. With a specialized graphical user interface, a scientist-user can construct a high-level visual specification that captures the essential computational dependencies in a scientific model. During the model construction process, the system uses its scientific domain knowledge to ensure that the model being built is consistent and coherent. The availability of background domain knowledge enables the system to automatically infer portions of the model specification that otherwise would be tedious for the user to specify. The final product of the construction process is an executable prototype of a scientific model. SIGMA accelerates the overall model-building process and eliminates the scientist's need to program in a formal language. Furthermore, the models developed with SIGMA are easier to understand and potentially reuse than typical low-level scientific modeling code. By providing an interactive model-building environment, SIGMA encourages users to experiment and try new approaches with a minimal investment of time. Thus SIGMA supports the essential trial-and-error character of the scientific modeling process. With its use of automated inference and other knowledge-based techniques, the SIGMA model-building environment achieves a significant advance over Fortran environments used in the scientific computation community today.

Acknowledgments

This research was co-funded by the Applied Information Systems Research Program in NASA' s Office of Space Science, and the Operations/Artificial Intelligence Program in NASA's Office of Advanced Concepts and Technology. We wish to thank the other members of the SIGMA Project for their intellectual contribution to the ideas expressed in this paper. These members include Michael Sims, David Thompson, Pandu Nayak, Christopher McKay, Jennifer Dungan, Yaron Gold, and Caitlin Griffith. Special thanks to Pandu for providing the RML representation language, for maintaining and augmenting the language to suit our needs, and for tutoring us in its use. Chris Merz, Justin Pogue, and Phil Liao also made significant contributions to the project effort. Finally, thoughtful and comprehensive comments by anonymous reviewers were of great assistance in improving this paper.

Appendix 1: Sample object and transform definitions

This appendix is intended to give the reader a better understanding of the representational structures in SIGMA's knowledge base. The first subsection includes object definitions for some of the objects referenced in the text and in Figure 17. Text describing the general structure of SIGMA's knowledge base appears within Section 5, and may be helpful in understanding these object definitions. Following this subsection are the definitions of the equations depicted in Figures 16, 18, and 19, along with a sample definition of a subroutine. The object definitions in the first subsection should be cross-referenced in understanding these transform definitions.

The syntax of the definitions in this appendix has been slightly altered for readability and some details have been omitted. However, the basic structure of the definitions remains intact. A symbol preceded by a "$" is a variable representing a quantity; A symbol preceded by a "&" is a variable representing an object.

A1 .1 Object definitions

```
(defobject physical-entity (modelled-object)  ;; physical-entity is a type of
                                              ;;  modelled-object, and inherits
                                              ;;  attributes from this object

                                              ;; Definitions of attributes that are
                                              ;;  local to physical-entity follow

  ((density
    :type density-quantity                    ;;  Type gives the object type that
                                              ;;  can be stored in this attribute
    :description "the mass per volume of the entity")
   (pressure
    :type pressure-quantity
    :description "the pressure of the entity")
   (temperature
    :type temperature-quantity
    :description "the temperature of the entity")
   (mass
    :type mass-quantity
    :description "the mass of the entity")

          etc...
  )
 :description "the collection of all things with physical
               properties")
```

```
(defobject radiation-interaction (atmospheric-object) ;; This object represents the
                                                       ;; collision between a radiation
                                                       ;; source and a physical entity

  ((refractivity                          ;; Refractivity and polarizability are
    :type refractivity-quantity           ;; quantities that are a function of both
                                          ;; the radiation source and the entity
    :description "the refractivity induced by the interaction")
   (polarizability
    :type polarizability-quantity
    :description "the polarizability induced by the interaction")
   (incident-radiation                     ;; a link to radiation source
    :type radiation-source
    :description "the radiation source involved in the interaction")
   (radiated-entity                        ;; a link to physical entity
    :type physical-entity
    :description "the physical entity being irradiated")
  )
 :description "the collection of interactions between radiation sources and
               physical entities")



(defobject force-field (field)
  ((first-body
    :type physical-entity)
   (second-body
    :type physical-entity)
   (force
    :type force-quantity)
   (length
    :type length-quantity))
)

(defobject mixture (physical-entity)  ;; A mixture is a specialization of
                                      ;;  physical-entity

  ((constituents                ;; This attribute is locally defined for mixtures
    :type mixture-constituent)  ;;  and stores an array of links to objects of
                                ;;  type mixture-constituent
   )
  :description "the collection of all mixtures")


(defobject mixture-constituent (physical-entity)  ;; A mixture-constituent is also
                                                  ;; a specialization of physical-entity
```

```
  ((mixing-fraction            ;; This attribute is locally defined for constituents
    :type mixing-ratio-quantity) ;;  and stores the percentage of this constituent
                                 ;;  with respect to the total amount of mixture
   )
  :description "the collection of all constituents of mixtures")



(defobject atmospheric-parcel (mixture)  ;; An atmospheric-parcel is a further
                                         ;;  specialization of mixture
  ((altitude          ;; a local attribute of atmospheric-parcel
    :type length-quantity)

   etc ...
   )
  :description "the collection of all atmospheric parcels")



;;; Following are objects related to the representation of quantities and
;;; scientific unit expressions

(defobject physical-quantity (modelled-object)

  ((value
    :type lisp-number)
   (unit
    :type scientific-unit-expression))

  :description "The set of all quantities")


(defquantity pressure-quantity          ;; Definition of pressure-quantity,
                                        ;;  an specialization of “physical-quantity”
  :description "Pressure is force per unit area."

  :derivatives ((distance dz-pressure))      ;; Derivative of pressure
                                             ;;  with respect to distance is a
                                             ;;  quantity called dz-pressure
  :unit pressure-unit)


(defunitype pressure-unit (factor-convertible-unit)
  ;;  Pressure is a measure of the force applied per unit area
  ;;  on a surface.
  ((definition (/ force-unit area-unit))))
```

```
(defunit bar pressure-unit   ;; bar is an instance of the class pressure-unit

  :documentation "The bar is a unit of pressure nearly equal to an
atmosphere."

  :SI-conversion-factor 1e5            ;; to covert a bar to the SI
                                       ;; standard unit, multiply by 1e5
  :conversion-factors (pascal 1d5
                  centibar 100
                  millibar 1000
                  atmosphere 0.986923d0))
```

A1.2 "Gravitational Force Equation "definition

```
(defequation gravitational_force_equation    ;;; See Figure 16

    :equation                          ;;; Equation formula:
    ("$F = $G * $m1 * $m2 / $r ** 2") ;;;  If the desired output variable is not
                                       ;;;  on the left hand side of the equation,
                                       ;;;  SIGMA calls REDUCE to rewrite the formula
                                       ;;;  in the proper form prior to execution

    :anchor-objects                    ;;; Definition of focal or "anchor" objects,
                                       ;;;  which serve as anchors for the equation
                                       ;;;  matching process.  All objects referenced
                                       ;;;  in the equation can be "derived" from
                                       ;;;  the anchors by following links.
                                       ;;;  In matching an equation, the anchor
                                       ;;;  objects are matched and bound to
                                       ;;;  instances first; then the
                                       ;;;  derived-object links are followed
                                       ;;;  to locate subsidiary instances.


    ((&ffield force-field))            ;;; &ffield is a local variable representing
                                       ;;;  an object of type "force-field"

    :derived-objects                   ;;; Definition of subsidiary objects,which are
                                       ;;;  derived from anchor object via links.

    ((&entity1 (first-body &ffield))
                                            ;;; &entity1 represents the object
                                            ;;;  linked to &ffield via the
                                            ;;;  "first-body" link.
     (&entity2 (second-body &ffield)))
                                            ;;; &entity2 is linked to the same
                                            ;;;  force-field as &entity1,
                                            ;;;  but via the "second-body" link.


    :variable-specs                    ;;; Definition of symbols in formula

    (($F (force &ffield))              ;;; $F is the "force" attribute of &ffield
     ($m1 (mass &entity1))
     ($m2 (mass &entity2))
     ($r (length &ffield)))

    :possible-output-variables         ;;; Only certain symbols are
```

```
                                    ;;; permitted as outputs
    ($F $m1 $m2 $r)

    :constants                      ;;; Definition of constant symbols

    (($G universal_gravitational_constant))

    :constraints                    ;;; Additional constraints on objects.
                                    ;;;  Note that constraint requiring
                                    ;;;  &entity1 and &entity2 to be distinct
                                    ;;;  is implicit. They are considered distinct
                                    ;;;  if two separate symbols are defined
                                    ;;;  in the derived-objects section above.
    ((= $r (center-distance &entity1 &entity2)))

    :citation mckay-pg-44           ;;; Pointer to a citation for the equation

    :description "Equation describing the gravitational force
       between two bodies with mass m1 and m2, respectively"
    )
```

A1.3 "Refractivity of a Gaseous Mixture" equation definition

```
(defequation refractivity_of_a_gaseous_mixture   ;;; See Figure 18

  :equation                              ;;; Equation formula
   ("$R = summation(&constits,  $N_i * $p_i)")

 :anchor-objects                         ;;; Definition of focal or "anchor" objects

   ((&mixture-interaction radiation-interaction)) ;; &mixture-interaction is a
                                                  ;;  local variable representing
                                                  ;;  a radiation-interaction object

 :derived-objects                 ;;; Definition of secondary objects derived
                                  ;;;  from anchor via links

   ((&mix mixture (radiated-entity &mixture-interaction))
    ;;; &mix represents the object that is connected to
    ;;;  &mixture-interaction via the radiated-entity link.
    ;;;  The specifier 'mixture' further constrains the object to be a mixture.
    ;;;  Without this specifier, the object is only constrained by the type
    ;;;  of the radiated-entity link, which is of type physical-entity.

    (&radiation (incident-radiation &mixture-interaction))
    ;;; &radiation represents the object that is connected
    ;;;  to &mixture-interaction via the incident-radiation link

    (&constits (constituents &mix) (forall &constits (phase &constits gaseous)))
    ;;; &constits is a list of all constituents of the mixture.
    ;;;  All of the constituents must be in gaseous phase.

    (&constit (one-of &constits))
    ;;; &constit is bound to a member of the list

    (&constit-interaction radiation-interaction
                 (and (radiated-entity &constit-interaction &constit)
                      (incident-radiation &constit-interaction &radiation)))
    ;;; &constit-interaction represents a radiation interaction object
    ;;;  such that the radiated-entity of &constit-interaction is &constit
    ;;;  and the incident-radiation of &constit-interaction is &radiation.
    ;;;  Note that &radiation is the incident-radiation of both
    ;;;  &constit-interaction and &mixture-interaction.


  :variable-specs     ;;; Definition of symbols in the formula

   (($R (refractivity &mixture-interaction)) ;; $R is the "refractivity" attribute
```

```
                                                ;;   of the &mixture-interaction object
 ($N_i (density &constit))
 ($p_i (polarizability &constit-interaction)))


:possible-output-variables  ;;; Symbols that are permitted as outputs

($R $N_i $p_i)

:citation mckay-pg-44       ;;; Reference to bibliographic citation for equation

:description "Equation describing how to calculate the refractivity of
               a gaseous mixture as a function of the density and polarizability
             of the gases in the mixture")
```

A1.4 "Density Computation" equation definition

```
(defequation density_computation   ;;; See Figure 19

  :equation                              ;;; Equation formula
   ("$N = $R / $p")

 :anchor-objects                         ;;; Definition of focal or "anchor" objects

   ((&interaction radiation-interaction))         ;; &interaction is a
                                                  ;;  local variable representing
                                                  ;;  a radiation-interaction object

 :derived-objects                  ;;; Definition of secondary objects derived
                                   ;;;  from anchor via links

   ((&entity (radiated-entity &interaction))
    ;;; &entity represents the object that is connected
    ;;;  to &interaction via the radiated-entity link

    (&radiation (incident-radiation &interaction))
    ;;; &radiation represents the object that is connected
    ;;;  to &interaction via the incident-radiation link

  :variable-specs     ;;; Definition of symbols in the formula

   (($N (density &entity))                   ;; $N is the "density" attribute
                                             ;;  of the &entity object
    ($R (refractivity &interaction))
    ($p (polarizability &interaction)))


   :possible-output-variables  ;;; Symbols that are permitted as outputs

   ($N $R $p)

   :citation mckay-pg-47       ;;; Reference to bibliographic citation for equation

   :description "Equation describing the refractivity (R) of a physical entity
                 as a function of its number density (N) and polarizability (p)")
```

A1.5 "Equation of State" subroutine definition

```
(defsubroutine equation_of_state

  :language fortran
  :subroutine                              ;;; Subroutine call
   ("eqstate($f_N2, $size_f_N2, $f_H2, $size_f_H2, $P, $size_P,
                 $N, $size_N, $T, $size_T")

 :anchor-objects                   ;;; Definition of focal or "anchor" objects
   ((&parcel atmospheric-parcel))

 :derived-objects                  ;;; Definition of secondary objects derived
                                   ;;;  from anchor via links

   ((&N2-constituent (constituents &parcel)
                       (molecular-composition &N2-constituent nitrogen-gaseous)))

    (&H2-constituent (constituents &parcel)
                       (molecular-composition &H2-constituent hydrogen-gaseous)))

  :variable-specs     ;;; Definition of symbols in the subroutine call

   (($f_N2 (mixing-fraction &N2-constituent)
           ($size_f_N2)     ;;; size of the array for this argument
                            ;;; (size = 1 if argument is a scalar)
           dimensionless)   ;;; no scientific units associated with argument
    ($f_H2 (mixing-fraction &H2-constituent)
           ($size_f_H2)
           dimensionless)
    ($P    (pressure &parcel)
           ($size_P)
           atmosphere)      ;;; P is expected in units of "atmosphere"
    ($N    (density &parcel)
           ($size_N)
           (/ mole liter))  ;;; N must be in units of mole/liter
    ($T    (temperature &parcel)
           ($size_T)
           kelvin))         ;;; T must be in kelvins

   :input-variables  ;;; Symbols that are subroutine inputs
                     ;;;  $T is (implicitly) the only output
   ($f_N2 $f_H2 $P $N)

    :description "Fortran subroutine to compute the temperature of an atmospheric
                 parcel composed of Nitrogen and Hydrogen, given the mixing fraction
                 of these gases and the pressure and density of the parcel")
```

Appendix 2: Experiment scenario description

This appendix provides the formal description for the Titan experiment scenario illustrated in Figure 3. This description has been simplified, and details have been omitted to improve the clarity of the presentation.

```
(defscenario lindal-scenario   ;;; Experiment scenario for Voyager-Titan encounter

  :objects                          ;;; Type definitions of objects in the scenario.
                                    ;;; All object variables have the form ?x

    ((atmospheric-parcel ?parcel)       ;;; ?parcel represents an object
                                        ;;;  of type atmospheric-parcel
     (parcel-constituent ?pc-n2)
     (parcel-constituent ?pc-h2)
     (radiation-interaction ?interaction-n2)
     (radiation-interaction ?interaction-h2)
     (radiation-interaction ?interaction))
              ;;; The radiation interaction objects represent the collision of the
              ;;; Voyager signal with the atmospheric parcel and its constituents

  :relations                            ;;; Defining relations between objects

    ((associated-planetary-body         ;;; The associated-planetary-body link
                    ?parcel titan)      ;;; in the ?parcel object is filled with
                                        ;;; the titan object.

     (components ?parcel (?pc-n2 ?pc-h2))  ;;; The components of ?parcel are the
                                           ;;; ?pc-n2 and ?pc-h2 objects

     (molecular-composition ?pc-n2 nitrogen-gaseous)  ;;; The molecular-composition
                                                      ;;; of ?pc-n2 is nitrogen-gaseous
     (molecular-composition ?pc-h2 hydrogen-gaseous)

            ;;; The following relations set up links in the radiation-interaction
            ;;;  objects so they point to the appropriate radiated-entity and
            ;;;  incident-radiation objects

     (radiated-entity ?interaction ?parcel)
     (radiated-entity ?interaction-n2 ?pc-n2)
     (radiated-entity ?interaction-h2 ?pc-h2)

     (incident-radiation ?interaction voyager-signal)
     (incident-radiation ?interaction-n2 voyager-signal)
     (incident-radiation ?interaction-h2 voyager-signal))

  :grid                             ;;; The object structure described above
```

```
                                    ;;; is repeated in an grid (or array) structure,
                                    ;;; with one structure at each altitude level

    (?parcel :stored-in (parcels titan) :axes (altitude))
       ;; This declares a grid of ?parcel object instances.
       ;;  The grid is stored in the 'parcels' slot of the 'titan' object instance.
       ;;  The 'titan' instance is already predefined as part of SIGMA's
       ;;  knowledge base.  This grid is indexed by the 'altitude' attribute
       ;;  of the stored ?parcel instances.

  :initial-values          ;;; The following values are initialized in the slots of
                           ;;;  instantiated objects.

    ((altitude ?parcel) (refractivity ?interaction)
     ;; The altitude slot of the ?parcel instance and the refractivity slot of the
     ;;  ?interaction instance are initialized.

     ;; Each pair below contains a value and its associated scientific unit.
     ;;  There are multiple sets of pairs because the quantities are part
     ;;  of the grid array structure; One value is initialized for each altitude.

         (((0.0 kilometer) (1.308e-9 dimensionless))
          ((0.5 kilometer) (1.284e-9 dimensionless))
          ((1.0 kilometer) (1.167e-9 dimensionless))
          ... etc))

)
```

References

H. Abelson, M. Eisenberg, M. Halfant, J. Katzenelson, E. Sacks, G. J. Sussman, J. Wisdom, and K. Yip (1989). Intelligence in Scientific Computing. Communications of the ACM, 32(5), 546-562.

S. Addanki, R. Cremonini, and J. S. Penberthy (1991). Graphs of models. Artificial Intelligence, 51(1-3), 145-177.

apE (1992). Software product. In Columbus, OH: Ohio Supercomputer Center.

W. Atwood, R. Blankenbecler, P. F. Kunz, B. Mours, and A. Weir (1990). The Reason Project No. SLAC-PUB-5242). Stanford Linear Accelerator Center.

AVS (1992). Software product. In Sunnyvale, CA: Stardent Computer, Inc.

D. Barstow (1985). Domain-Specific Automatic Programming. IEEE Transactions on Software Engineering, SE-11(11), 1321-1336.

D. Barstow, R. Duffey, S. Smoliar, and S. Vestal (1982). An Overview of Φnix. In National Conference on Artificial Intelligence, (pp. 367-369). Pittsburgh, PA: AAAI Press.

A. Borning (1981). The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. ACM Transactions on Programming Languages and Systems, 3(4), 357-387.

B. W. Char (1988). Maple Users Guide. Waterloo, Ontario: WATCOM Publications.

G. O. Cook Jr. (1990). ALPAL, A Program to Generate Physics Simulation Codes from Natural Descriptions. International Journal of Modern Physics C, 1(1), 1-51.

A. L. Davis, and R. M. Keller (1982). Data Flow Program Graphs. IEEE Computer(February), 26-41.

J. L. Dungan, and R. Keller (1991). Development of an Advanced Software Tool for Ecosystem Simulation Modelling (abstract). Bulletin of the Ecological Society of America, 72(2), 104.

Extend (1992). Software product. In San Jose, CA: Imagine That, Inc.

B. Falkenhainer, and K. Forbus (1991). Compositional Modeling: Finding the right model for the job. Artificial Intelligence, 51(1-3), 95-143.

R. E. Filman (1988). Reasoning with Worlds and Truth Maintenance in a Knowledge-Based Programming Environment. Communications of the ACM, 31(4), 382-401.

B. Ford, and F. Chatelin (Ed.). (1987). Problem Solving Environments for Scientific Computing. North-Holland.

J. W. Forrester (1961). Industrial Dynamics. Cambridge: MIT Press.

E. Glinert (Ed.). (1990). Visual Programming Environments. Los Alamitos: IEEE Computer Society Press.

R. V. Guha (1991) Contexts: A formalization and some applications. Ph.D., Stanford University Computer Science Department.

R. V. Guha, and D. B. Lenat (1990, Cyc: A Mid-Term Report. AI Magazine, p. 32-59.

A. C. Hearn (1991). REDUCE User's Manual, Version 3.4 No. CP78). RAND Corporation, Santa Monica, CA.

A. C. Hindmarsh (1983). ODEPACK, A Systematized Collection of ODE Solvers. In R. S. Stepleman (Eds.), IMACS Transactions on Scientific Computation (pp. 55-64). Amsterdam: North-Holland.

Iconicode/IDF (1992). Software product. In Palo Alto, CA: Iconicon.

IDL (1992). Software product. In Boulder, CO: Research Systems, Inc.

N. Iscoe (1991). Domain Modeling -- Evolving Research. In Sixth Annual Knowledge-Based Software Engineering Conference, (pp. 234-236). Syracuse, NY: IEEE Computer Society Press.

W. L. Johnson, and D. R. Harris (1991). Sharing and Reuse of Requirements Knowledge. In Sixth Annual Knowledge-Based Software Engineering Conference, (pp. 57-66). Syracuse, NY: IEEE Computer Society Press.

E. Kant, F. Daube, W. MacGregor, and J. Wald (1991). Scientific Programming by Automated Synthesis. In M. R. Lowry & R. D. McCartney (Eds.), Automating Software Design Menlo Park: AAAI Press.

R. M. Keller (Ed.). (1992). Proceedings of the AAAI Workshop on Automating Software Design (Theme: Domain-Specific Software Design). AI Research Branch, NASA Ames Research Center, Moffett Field, CA.

Khoros (1992). Software product. In Albuquerque, NM: Khoros Consortium, EECE Dept., Univ. of New Mexico.

H. J. Kook, and G. S. Novak Jr. (1991). Representation of Models for Expert Problem Solving in Physics. IEEE Transactions on Knowledge and Data Engineering, 3(1).

LabVIEW (1992). Software product. In Austin, TX: National Instruments.

G. F. Lindal, G. E. Wood, H. B. Hotz, D. N. Sweetman, V. R. Eshelman, and G. L. Tyler (1983). The atmosphere of Titan: An analysis of the Voyager 1 radio occultation measurements. Icarus, 53, 348-363.

R. Ling, and L. Steinberg (1992). Automated Modeling in Computational Heat Transfer. In E. Kant (Ed.), AAAI Fall Symposium on Intelligent Scientific Computation, (pp. 68-73). Cambridge, MA: AAAI.

MATLAB (1989). Software product. In Natick, MA: The MathWorks, Inc.

C. P. McKay, J. B. Pollack, and R. Courtin (1989). The Thermal Structure of Titan's Atmosphere. Icarus, 80, 23-53.

P. P. Nayak (1992) Automated Modeling of Physical Systems. Ph.D., Stanford University.

R. S. Palmer, and J. F. Cremer (1991). SIMLAB: Automatically Creating Physical Systems Simulators (Technical Report No. TR-91-1246). Department of Computer Science, Cornell University.

D. Robertson, A. Bundy, R. Muetzelfeldt, M. Haggith, and M. Uschold (1991). Eco-Logic: Logic-Based Approaches to Ecological Modelling. Cambridge: MIT Press.

D. Robertson, A. Bundy, M. Uschold, and R. Muetzelfeldt (1989). The ECO Program Construction System: ways of increasing its representational power and their effects on the user interface. International Journal of Man-Machine Studies, 31, 1-26.

S. W. Running, and J. C. Coughlan (1988). A General Model of Forest Ecosystem Processes for Regional Applications: I. Hydrologic Balance, Canopy Gas Exchange and Primary Production Processes. Ecological Modelling, 42, 125-154.

M. Spenke, and C. Beilken (1990). An Overview of GINA -- the Generic Interactive Application. In D. A. Duce (Ed.), User Interface Management and Design, . Lisbon: Springer-Verlag.

STELLA/IThink (1992). Software products. In Lyme, NH: High Performance Systems.

D. S. Weld (1990). Approximation reformulations. In Eighth National Conference on Artificial Intelligence, (pp. 407-412). Cambridge, MA: AAAI Press.

D. S. Weld, and J. deKleer (Ed.). (1990). Readings in Qualitative Reasoning about Physical Systems. Palo Alto: Morgan Kaufmann.

S. Wolfram (1988). A System for Doing Mathematics by Computer. Redwood City: Addison-Wesley.