Introduction

Science-o-matic is an integrated environment for scientific computation, reasoning, induction, abduction and visualization. Its primary tool is a general-purpose deductive reasoner for complex scientific applications called the Scientific Metareasoner (or just “Metareasoner”). The Metareasoner is in turn built upon Scilog, a scientific reasoning language that extends the core of Prolog with special predicates for reasoning with processes and states, and by associating and manipulating meta-data about scientific values. The Metareasoner combines five Scilog components for representing metaphysical facts, scientific theory, scientific laws, scientific data and analytical rules.

The Metareasoner’s architecture supports scientific reasoning in two manners. First, it supports a variety of types of scientific reasoning using just one knowledge base. This is done by calling the different components in different orders. For example, calling the data component earlier favors reading observations while calling the theory component earlier favors reasoning from first principles. Secondly, while it is designed to support scientific reasoning (with the theory, law and data components), it also supports mathematical, definitional and “common-sense” reasoning with the metaphysical and analytical components.

Additionally, although Metareasoner only reasons, its architecture supports discovery in two ways. First, the existence of three scientific components allows scientists to precisely specify the knowledge to add or change. New observations go in the data component; new high-level assertions go in the theory component; and new mid-level assertions bridging high-level to their observable manifestations go in the law component. An example from classical physics might be Kepler's Laws bridging Newton's the and Brahe's data. Second, this three component architecture allows the act of finding better connections between data and laws, and between laws and theory. Such an approach is outlined in [Phillips 2001].

Another advantage of defining the Metareasoner in terms of Scilog is that the Metareasoner supports a higher-level programming interface called Scilog High Level Language (or just “High Level Language”). Though Scilog's definition is sufficiently precise for scientific applications, it is an annoying low-level programming tool. The Metareasoner allows graphical user interface (GUI) support for scientific programming. For example, instead making scientists edit ASCII Scilog files, High Level Language lets them:

- see and edit the hierarchy of the ontology of objects as a Java JTree object,
- see and edit scientific meta-data objects (e.g. dimensions, units and domains) and their properties with object-specific pull-down menus,
- use more intuitive infix notation for equations, and
• see and edit the decision trees as Java JTree objects.

This manual is organized as follows.

Chapter 1 gives a brief motivating Tutorial Introduction.
Chapter 2 describes the Science-o-matic Models and Ontologies.
Chapter 3 describes Values and Facts.
Chapter 4 discusses the specification of Static Knowledge (Equations and Decision Trees not associated with Process Classes).
Chapter 5 discusses Process Classes
Chapter 6 describes Simulations
Chapter 7 discusses General Inductive and Abductive Tools
1 Tutorial Introduction

To get started go to the tutorial directory and type the following command.

Figure 1-1: Starting Science-o-matic:

```
unix> meta_reasoner -gui tutorial.model
```

This loads the model `tutorial.model` into the Science-o-matic and tells it to use the GUI interface which looks like the figure below:

Figure 1-2: Starting Science-o-matic:

Suppose we want to compute the mass of some simple homogeneous object. If we know the object's volume and the density of the material of which it was made they were made we could use the simple formula:

**Equation 1-1: Mass is the product of density and volume:**

\[
\text{mass} = \text{density} \times \text{volume}
\]

To see this equation go to menu item **Edit Knowledge Component** and then choose **Theory**. Then, under **Equation(s)** choose **density_eqn**. Doing so brings up the following window:

Figure 1-3: The equation `density_eqn`: 
The equation name is given as the first parameter to the `equation` term. The equation has one entity (the object whose mass, density and volume is under consideration). This entity is represented by entity variable `object`. This entity variable may stand-in for any member of class `physical_object_class` and this fact is represented by the text:

**Figure 1-4: The entity variable `object` of equation `density_eqn`:**

```
object instance_of physical_object_class
```

in the list in the second parameter. The third parameter states that for this equation to be applicable the object must have uniform density. The actual equation is given last and says that `object`'s `mass_attr` is equal to its `volume_attr` times its `objects_material_density_attr`. We will revisit this equation in chapter 4.

Of course to use this equation we need to know an object's volume and the density of its material. These are data on particular objects. Go to **Edit Knowledge Component** and then choose **Data**. Under **Property** press **Edit**. The following window should appear:

**Figure 1-5: The Data Component:**
The top left pane gives the ontology of objects. Choose the lead_weight object by clicking on empirical_entity_class, then physical_object_class, then dropable_object_class, then solid_dropable_object_class, and finally lead_weight. After doing so when clicking on the top right pane you should see the following options:

Figure 1-6: The Data Component Properties of lead_weight:

<local properties>
objects_material_attr=lead
volume_attr=value(4,default_ml_domain)

Object lead_weight has no inheritable properties (it is not a class) but two local properties that give both its material and the volume. Additionally, lead_weight inherits all inheritable properties of all classes everything down solid_dropable_object_class. Click on solid_dropable_object_class, go to the top right pane and you'll see the inheritable property:

Figure 1-7: The Data Component Properties of solid_dropable_object_class:

objects_material_phase_attr=common_solid

To finally compute the object's mass we need its density given that its material is lead in the “common solid” phase. Go to Edit Knowledge Component and then choose
Law. Under D-Tree(s) choose `material_density_dtrees`. The following should appear:

**Figure 1-8: The Decision Tree material_density_dtrees:**

The decision tree is given in the left pane. Under `object.objects_material_attr` click `==(lead)`, then on `object.objects_material_phase_attr`, then on `==(common_solid)`. You will see that the density value is given at the leaf as `value(11.35,default_grams_per_ml_domain)`.

So now we see that we have all of the knowledge needed to compute the mass of `lead_weight`. To do so go to **Determine Value** and then **Determine**. You should see this:

**Figure 1-9: The Value-Determining Window:**
The first option specifies the order in which the knowledge components are queried. Leave that as theorize.

Next you will need to tell it the object you are interested in. Either type lead_weight in the Object text area or click on empirical_entity_class, then physical_object_class, then dropable_object_class, then solid_dropable_object_class, and finally lead_weight.

The attribute comes next. Either type mass_attr in the Attribute text area or click on attribute_class, then empirical_attribute_class, then object_measure_attribute_class, then mass_attribute_class, and finally mass_attr.

Leave the State or Series text area blank and press Determine value. You should see the following window:

**Figure 1-10: The Response Window:**

which yields our answer of 4 ml times 11.35 grams per ml equals 45.4 grams.
2 The Model and Ontology

In the previous chapter we saw how the Science-o-matic can be queried about knowledge in its ontology. In this chapter we will discuss models and the ontology more thoroughly.

2.1 Models

Model files tell which Scilog files go into which component. By convention model files have extension .model. The tutorial model file contains the following text:

**Figure 2-1: The Tutorial Model tutorial.model:**

```
metaphysics ("std.ontol.pro",
             "std.back.pro",
             "tutorial.ontol.pro",
             "tutorial.back.pro"
          )
theory    ("density_eqn.pro"
          )

law       ("material_density_dtree.pro"
          )
data       ("tutorial_data.pro"
          )
analytics ("std.math.pro",
          "calculus.pro"
          ).
```

Model files may have up to five sections (metaphysics, theory, law, data and analytics), one for each component. Sections may be listed in any order but the Science-o-matic always loads them in the order metaphysics, theory, law, data and analytics. Each section gives the component name followed by a parenthesized, comma-separated list of double-quoted Scilog filenames. The file ends with a period ("."). The C-style comments /* ... */ may be used in model files.

The knowledge that goes into each of the components is detailed below.

1. **metaphysics.** This component stores four types of background knowledge:
   (a) the ontology,
   (b) definitions of some standard Prolog predicates,
   (c) attributes of background concepts, and
   (d) limitations on the model.

   The ontology is discussed in the next subchapter while the other three are discussed in Chapter _, Architecture and Reasoning.

2. **theory.** This component gives high level (“theoretical”) domain knowledge. The file `density_eqn.pro` specifies an equation that tells to compute the mass of an object
from its density and volume.

3. **law.** This component gives intermediate level domain knowledge. The file material_density_dtree.pro specifies a decision tree that gives the density of an object given the type of material and the phase of the material of which it is made.

4. **data.** This component gives low level data. The file tutorial_data.pro contains specific data about domain objects like lead_weight's material and material phase.

5. **analytics.** This component tells how to recast problems. The file std.math.pro gives an equation that defines random numbers for all objects and gives several equations that inter-relate a state's beginning time, ending time and duration. It also tells what attribute is obtained when another attribute is integrated (e.g. velocity results from integrating acceleration with respect to time). The file calculus.pro has the rules to do differential an integral calculus so this knowledge can be applied.

2.2 The Ontology

The ontology defines the relationships among all concepts known by the reasoner in an is-a hierarchy. It is given in metaphysical files with extension .ontol.pro, which must have all and only is_a(subclass, superclass) and instance_of(instance, class) predicates. In the figure above the file std.ontol.pro gives the definitions of standard domain-independent Scilog objects (e.g. standard non-empirical attributes) while the file tutorial.ontol.pro domain-specific objects. A portion of tutorial.ontol.pro is given in Figure 2-2:

**Figure 2-2: A Portion of File tutorial.ontol.pro:**

```pro
is_a(dropable_object_class, physical_object_class).
is_a(solid_dropable_object_class, dropable_object_class).
is_a(liquid_dropable_object_class, dropable_object_class).
instance_of(lead_weight, solid_dropable_object_class).
instance_of(latex_ball, solid_dropable_object_class).
instance_of(water_drop, liquid_dropable_object_class).
```

Among other things this file defines dropable_object_class as a subclass of predefined class physical_object_class, defines solid_dropable_object_class as a subclass of dropable_object_class, and defines lead_weight as an instance of solid_dropable_object_class.

The file std.ontol.pro defines several importance classes of objects. Under everything, the root of the ontology, there are four main classes:

1. **empirical_entity_class.** This class represents everything the Universe that science could claim as subject matter, all empirically describable things. Important subclasses include physical_object_class (e.g. lead weights), physically_manifested_patterns_class (e.g. materials like lead and phases of matter like “common solids”),
physical_object_relationship_class (e.g. the lead weight being in someone's hand), and process_class (e.g. actions like lead weights falling).

2. attribute_class. This class represents all attributes, or mappings from things to their properties. Important subclasses include empirical_attribute_class (all attributes that describe empirical properties, like mass, volume, etc. and meant to describe instances of empirical_object_class), cultural_convention_attribute_class and program_operation_attribute_class that are meant to describe members of cultural_convention_class and program_operation_class respectively.

3. cultural_convention_object_class. This class holds objects defined by cultural convention. Important subclasses include dimension_class (e.g. length), units_class (e.g. meters), state_class (that specify the times of events) and domain_class (that specify the precision, dimension, units and upper and lower bounds of measurements).

4. program_operation_object_class. This class holds objects relevant to the running of the program. Important subclasses include data_type_class and operator_class (e.g. addition, subtraction).

More pre-defined classes are presented in Appendix I: Pre-defined Classes.

2.3 The Syntax of Atoms

The Science-o-matic language is built on Scilog, and Scilog in turn is based on Prolog. The language complies with Prolog's conventions for atoms. In particular:

1. All names are case sensitive and must begin with a lowercased letter. Thereafter they may contain letters in either case, digits and the underscore character _. Names may be up to 95 characters long.

2. Integers must be between -2147483648 and 2147483647 inclusive. Floating points must have an absolute value between -1.79769e-308 and 1.79769e+308.

2.4 Internal Prolog Syntax

This internally-used default form is used in files ending with extension .pro. In particular:

1. The primary representation of all terms conforms to Prolog's syntax for terms of name ( <comma_separated_argument_list> ). We have seen this with the density_eqn: equation(density_eqn,<list1>,<list2>,expression).

2. Strings are delimited by the single quote ' (the apostrophe character).

3. Lists are delimited by square brackets [ and ]. They may have zero or more comma-separated items (names, terms, numbers, strings or other lists) in them.

4. C-style comments of arbitrary length are delimited between /* and */. To-the-end-of-the-line style comments begin with the percent sign %.
3 Values and Facts

3.1 Facts

Scilog use the frame system `<object, attribute, value>` tuple to store and query knowledge. This means that the thing `object` has some property `attribute` whose value is `value` (e.g. `<lead_weight, mass_attr, 45.4 grams>`). Although Scilog fully supports querying for values given the object and attribute, there it has limited ability to query for unbounded objects and attributes.

Prolog ground facts are meant for measurements, statements of cultural convention and other direct assertions of knowledge for which no further justification is warrented. They are to be given in tables, or are to be associated with individual objects or classes.

3.2 Values

A lot must be known to work with data effectively. For example, units and dimensionality are two important pieces of metadata that, when not dealt with properly, have caused dramatic and costly system failures [ref].

Scilog addresses this by associating several pieces of metadata with values. Values have:

1. a **primary value** (e.g. 45.4)
2. a **certainty range** (e.g. 0.05)
3. an object being described (e.g. lead_weight)
4. an aspect of the object being described (e.g. mass_attr)
5. a **state** telling (among other things) when this value holds (e.g. universal_state). States will be described later.
6. the primary value's **domain**, which includes its dimensionality (mass), its units (grams), its precision, and its set of legal values (mass must be non-negative).

Primary values that have one or more pieces of such metadata are called **annotated values**.

Scilog uses the `value` function to define annotated values.

**Figure 3-1: Examples of Defining Numeric Annotated Values Using value:**

The `value` function may take a variable number of parameters. The first parameter states the primary value and must be present. One option is to have a number as the primary value. If this is done then that primary value is meant as the mean of the range. The single numeric primary value may be followed by a second number that tells the half width of the certainty range. (A certainty range of a value is the range, centered around the mean, that includes the true value with some given probability. For example, (45.4, 0.05) means 45.4±0.05. The half width in this case is 0.05.) Any additional arguments must be concepts that tell the domain of the value (as described in __), the state of the value (as described in __), the object being described and the attribute of that object.
These additional conceptual descriptors may be in any order but if the object being described is itself a domain, state or attribute then the describing domain, state or attribute must be given before the object.

Sampled numeric annotated values are specified in a similar fashion. The primary value is a non-empty Prolog list of numbers. Sampled values lack a certainty range (the range being implicit in the sampling). Although an ascending order is highly recommended the numbers may be in any order. Also, as they are all considered equally-probable, the numbers may be repeated.

The value function may also be used when the primary value is a concept:

**Figure 3-2: Example of Defining a Symbolic Annotated Value Using value:**

This value states that, of all members of __.

There is also a notion of certainty ranges for concepts that is similar to the certainty ranges of floating point numbers. If the term member_of is given as the second argument for concept primary values then the resulting annotated value is taken to be a non-specific member of the concept class. That is the concept identifies some instance of the give class but does not tell which one. Reasoning can still be done with such non-specific values using inherit clauses:

**Figure 3-3: Example usage of non-specific concept reasoning:**

This query asks what the phase of matter of any member of class solid_dropable_object_class.

3.3 Attributes

Attributes allow some aspect of some object to be described.

3.4 Domains, Dimensions, Units and Limits
3.4.1 The Basics of Dimensions and Units

Our system knows some attributes of several objects, but say we made a mistake. Say we accidentally said that the value for lead_weight was volume_attr was -4.00 instead of 4.00 ml. We might therefore make incorrect inferences, like using the equation density=mass/volume to compute the value of -45.4 grams for mass_attr instead of 45.5 grams. Obviously we need to give Scilog an idea of what values are legal, and when.

Domains address this issue. Domains specify an attribute's legal set of values (including its datatype), its units (including its dimension), its measuring instrument-determined precision information and the types of objects that can be described. They can be used to detect illegal values as they are computed.

There are three numeric domain datatypes (integer, float_pt and fixed_pt) and one non-numeric one (concept). Numeric domains have units that are specified by the attribute units_attr. Units in turn have dimensions that are specified by the attribute units_dimension_attr.

If no domain is given for a value that could be interpreted as an integer then its domain defaults to default_integer_domain with units unitless. If no domain is given for a value that must be interpreted as a floating point number then its dimension defaults to default_float_pt_domain with units unitless. Units unitless has dimension dimensionless.

During unification Scilog automatically converts values with different units but the same dimension. Unification of values with different dimensions, however, always fails.

(Example)

Units and dimensions impact the usage of trigonometric functions. These functions may only use value with angular domains.

(Example)

All dimensions are instances of dimension_class. The definitions of dimensions are obtainable from Edit Metaphysical Object -> Dimensions/Units. This brings up a window that displays (and edits) dimensions and their units.

The biggest distinction is between atomic dimensions (which are not defined in terms of any other dimension) and composite ones (which are). For example, length is an atomic dimension with units meters, km, dm, cm and mm.
Units **meters** is the default measure of **length**. Other units are defined in terms of how many of the other units would equal exactly 1 meter.

Composite dimensions are similar except that their definition also includes the atomic dimensions (and the powers) from which they are composed. For example, force is equal to length to the first power, times mass to the first power, times time to the negative second power.

### 3.4.2 Advanced Dimensions and Units

Dimensions are described by the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_dim_atomic_attr</td>
<td>Maps to {true, false}. Is true if the dimension is not defined in terms of other dimensions and is false otherwise.</td>
</tr>
<tr>
<td>dim_main_units_attr</td>
<td>Maps to a member of <strong>units_class</strong>. Tells the preferred units of the dimension.</td>
</tr>
<tr>
<td>dim_component_attr</td>
<td>(For dimensions with is_dim_atomic_attr equal to false.) Maps to a “dimension component object”. Such objects have two single-valued attributes. dim_component_dim_attr tells an atomic dimension of which the described dimension is (partially) defined. dim_component_power_attr tells a unitless numeric power to which the dim_component_dim_attr dimension is raised.</td>
</tr>
</tbody>
</table>

All units are instances of **units_class**. Units are described by the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>units_dimension_attr</td>
<td>Maps to a member of <strong>dimension_class</strong>. Tells the dimension that the units describe.</td>
</tr>
<tr>
<td>units_convert_attr</td>
<td>(For the dim_main_units_attr units of atomic dimensions.) Maps to a “units conversion object”. Such objects have three single-valued attributes. units_convert_mult_attr maps to a unitless number that tells what to multiply the value in the main (“from”) units by to obtain the equivalent value in “to” units. (E.g. for <strong>meters_to_km</strong> it is 0.001, for <strong>meters_to_cm</strong> it is 100.) Optional from_units_attr tells the units being converted from (just the main units themselves). to_units_attr tells the units being converted to.</td>
</tr>
</tbody>
</table>
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>units_component_attr</td>
<td>(For units of non-atomic dimensions.) Maps to a “units component object”. Such objects have two single-valued attributes. units_component_units_attr maps to a units of the atomic dimension. units_component_power_attr maps to a unitless number that tells how much to raise the atomic units to to obtain the non-atomic units.</td>
</tr>
</tbody>
</table>

3.4.3 Basics of Domains and Limits

While dimensions prevent Scilog from confusing 1 second from 1 meter, and make Scilog equate 1 meter with 100 cm, they do not tell Scilog that -1 meter is an illegal length. Domains exist to assign units, value ranges, datatypes and precision information to values.

All domains are instances of `domain_class`. The definitions of domains are obtainable from **Edit Metaphysical Object -> Domains**. This brings up a window that displays (and edits) domains.

The example above shows the information for domain `default_celsius_domain`, which is an instance of both `temperature_domain_class` and `default_float_pt_single_precision_domain_class`, which in turn are (proper) subsets of `domain_class`. Domain `default_celsius_domain` is a floating point domain with mantissa and exponent sizes 23 and 8 bits respectively. Its lowest legal value is -273.15 and highest is approximately $1.7014 \times 10^{38}$.

Domains are a rare instance where multiple inheritance is encouraged. Many domains like `default_celsius_domain` are instances of a dimension-specifying class (in this case `temperature_domain_class`) and a precision-specifying class (in this case `default_float_pt_single_precision_domain_class`). This
arrangement cleanly separates the specifications of units from that of precision.

3.4.4 Advanced Domains and Limits

Numeric domains also have the following upper and lower bounds:

1. **lo_range_define_limit** and **hi_range_define_limit**: The range is logically defined to have these hard endpoints. Example: masses cannot be negative. **lo_range_define_limit** defaults to the C language's value for -DBL_MAX on the implemented architecture (often -1.79769e+308) if it is not explicitly specified. Similarly, **hi_range_define_limit** defaults to the C language's value for DBL_MAX on the implemented architecture (often 1.79769e+308) if it is not explicitly specified. The range $[lo\_range\_define\_limit..hi\_range\_define\_limit]$ is called **range_define_limit**.

2. **lo_system_limit** and **hi_system_limit**: The system being measured has these physical endpoints. Example: __. These limits default to the range limits if not specified. The range $[lo\_system\_limit..hi\_system\_limit]$ is called **system_limit**.

3. **lo_detect_limit** and **hi_detect_limit**: Recording instruments cannot measure values outside of this range. Example: __. These limits default to the range limits if not specified. The range $[lo\_detect\_limit..hi\_detect\_limit]$ is called **detect_limit**.

4. **lo_saturate_limit** and **hi_saturate_limit**: The system empirically does not exhibit many values outside this range, even though (in principle) it could. Example: __. These limits default to the system limits if not specified. The saturate limits are **soft limits**; are not meant to be a definitive threshold. The range $[lo\_saturate\_limit..hi\_saturate\_limit]$ is called **saturate_limit**.

5. **lo_reliable_limit** and **hi_reliable_limit**: Instruments cannot be relied upon to reproducibly to detect values outside this range. Example: __. These limits default to detect limits. The reliable limits are **soft limits**; are not meant to be a definitive threshold. The range $[lo\_reliable\_limit..hi\_reliable\_limit]$ is called **reliable_limit**.

6. **lo_observed_limit** and **hi_observed_limit**: No values were observed beyond these values. These limits default to the more restrictive of the detect limits and system limits if not specified or computed. The range $[lo\_observed\_limit..hi\_observed\_limit]$ is called **observed_limit**.

Three classes of consistency constraints use these domains: system constraints, instrument constraints and data constraints. Their definitions are given in the table below.
### Name | Constraint
--- | ---
System | $\text{saturate\_limit} \subseteq \text{system\_extent} \subseteq \text{range\_define\_limit}$
Instrument | $\text{reliable\_limit} \subseteq \text{detect\_extent} \subseteq \text{range\_define\_limit}$
Data | $\text{observed\_limit} \subseteq \text{system\_extent}$  
 | $\text{observed\_limit} \subseteq \text{detect\_extent}$

Scilog uses this meta-knowledge in three ways:

1. **Consistency checking**: Scilog can check values generated during computation to see if they violate any of the system, instrument or data constraints. Users specify whether violations halt execution or generate log file reports. Section 8.3.1 tells which domain inconsistency checks exist and how to either log or halt execution upon their detection.

2. **Dimensionality checking**: Values with different dimensions will fail to unify. For example, if domain `meter\_domain` has dimension `length` and domain `second\_domain` has dimension `time` then their values can not be unified.

3. **Automatic unit conversion**: Values with the same dimension but different units will have their units converted for comparisons and numeric operations.

In addition, the following datatypes have these addition machine precision parameters:

- **float\_pt** domains may specify the length of the mantissa (`float\_mantissa`) and exponent (`float\_exponent`) fields in binary digits. `float\_pt` values either have:
  1. an assumed Gaussian distribution and certainty bars that they have a user-specified probability of being within, or,
  2. some number of randomly chosen sample values that define a composite value.

- **fixed\_pt** domains may specify the delta (`fixed\_delta`) that is the constant increment between successive domain values.

- **concept** domains may specify a class to which values must belong. Non-negative integer `class\_size` can be associated with the class and tells its number of members. Concept values may also just specify membership in a class instead of giving a particular instance. This implements the conceptual equivalent of certainty ranges.

The following domains, units and dimensions are predefined in Scilog. In general the MKS system is followed.
<table>
<thead>
<tr>
<th>Dimension</th>
<th>Definition</th>
<th>Units</th>
<th>Domains(^1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimensionless</td>
<td>(atomic)</td>
<td>unitless</td>
<td>default_integer_domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(min = C's INT_MIN, max = C's INT_MAX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>count_domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(min = 0, max = C's INT_MAX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default_float_pt_domain</td>
</tr>
<tr>
<td>length</td>
<td>(atomic)</td>
<td>meters</td>
<td>default_meters_domain (min = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default_position_domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>km</td>
<td>default_km_domain (min = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cm</td>
<td>default_cm_domain (min = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mm</td>
<td>default_mm_domain (min = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>microns</td>
<td>default_microns_domain (min = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nm</td>
<td>default_nm_domain (min = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>angstroms</td>
<td>default_angstroms_domain (min = 0)</td>
</tr>
<tr>
<td>time</td>
<td>(atomic)</td>
<td>seconds</td>
<td>default_seconds_domain</td>
</tr>
<tr>
<td>angle</td>
<td>(atomic)</td>
<td>degrees and</td>
<td>default_0_to_360_degrees_domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>radians</td>
<td>(min = 0, max = 360)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default_neg_180_to_180_degrees_domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(min = -180, max = 180)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default_0_to_2pi_radians_domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(min = 0, max = 2\pi)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default_neg_pi_to_pi_radians_domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(min = -\pi, max = \pi)</td>
</tr>
<tr>
<td>mass</td>
<td>(atomic)</td>
<td>grams</td>
<td>default_grams_domain (min = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default_kg_domain (min = 0)</td>
</tr>
<tr>
<td>charge</td>
<td>(atomic)</td>
<td>coulombs</td>
<td>default_coulombs_domain</td>
</tr>
<tr>
<td>temperature</td>
<td>(atomic)</td>
<td>kelvin</td>
<td>default_kelvin_domain (min = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default_celsius_domain (min = -273.15)</td>
</tr>
<tr>
<td>area</td>
<td>length(^2)</td>
<td>meters_sqr</td>
<td>default_meters_sqr_domain (min = 0)</td>
</tr>
<tr>
<td>volume</td>
<td>length(^3)</td>
<td>liters</td>
<td>default_liters_domain (min = 0)</td>
</tr>
<tr>
<td>frequency</td>
<td>time(^{-1})</td>
<td>hertz</td>
<td>default_hertz_domain (min = 0)</td>
</tr>
<tr>
<td>velocity</td>
<td>length * time(^{-1})</td>
<td>meters_per_sec</td>
<td>default_meters_per_sec_domain</td>
</tr>
</tbody>
</table>

\(^1\) All listed domains have all upper limits that default to the implemented C definition of DBL_MAX. All domains have lower limits that default to the implemented C definition of -DBL_MAX except for default_kelvin_domain (0), default_celsius_domain (-273.15), and the domains for length, mass, area, volume, frequency, density, energy and pressure (all 0). The one exception to the previous sentence is default_position_domain (-DBL_MAX).
<table>
<thead>
<tr>
<th>Dimension</th>
<th>Definition</th>
<th>Units</th>
<th>Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>acceleration</td>
<td>length * time^2</td>
<td>meters_per_sec_sqr</td>
<td>default_meters_per_sec_sqr_domain</td>
</tr>
<tr>
<td>angular_velocity</td>
<td>angle * time^-1</td>
<td>revolves_per_sec</td>
<td>default_revolves_per_sec_domain</td>
</tr>
<tr>
<td>angular_acceleration</td>
<td>angle * time^2</td>
<td>revolves_per_sec_sqr</td>
<td>default_meters_domain</td>
</tr>
<tr>
<td>density</td>
<td>length^-1 * mass</td>
<td>kg_per_l</td>
<td>default_kg_per_l_domain (min = 0)</td>
</tr>
<tr>
<td>momentum</td>
<td>length * mass * time^-1</td>
<td>kg_m_per_sec</td>
<td>default_kg_m_per_sec_domain</td>
</tr>
<tr>
<td>force</td>
<td>length * mass * time^-2</td>
<td>newtons</td>
<td>default_newtons_domain</td>
</tr>
<tr>
<td>energy</td>
<td>length^2 * mass * time^-2</td>
<td>joules</td>
<td>default_joules_domain (min = 0)</td>
</tr>
<tr>
<td>power</td>
<td>length^2 * mass * time^-3</td>
<td>watts</td>
<td>default_watts_domain (min = 0)</td>
</tr>
<tr>
<td>pressure</td>
<td>length^-1 * mass * time^-2</td>
<td>pascals</td>
<td>default_pascals_domain</td>
</tr>
<tr>
<td>current</td>
<td>charge * time^-1</td>
<td>amps</td>
<td>default_amps_domain</td>
</tr>
<tr>
<td>electric_field</td>
<td>length * mass * charge^-1 * time^-1</td>
<td>volts_per_meter</td>
<td>default_volts_per_meter_domain</td>
</tr>
<tr>
<td>electric_potential</td>
<td>length^2 * mass * charge^-1 * time^-2</td>
<td>volts</td>
<td>default_volts_domain</td>
</tr>
<tr>
<td>resistance</td>
<td>length^2 * mass * charge^-2 * time^-4</td>
<td>ohms</td>
<td>default_ohms_domain (min = 0)</td>
</tr>
<tr>
<td>capacitance</td>
<td>length^2 * mass^-1 * charge^2 * time^-1</td>
<td>farads</td>
<td>default_farads_domain</td>
</tr>
<tr>
<td>magnetic_field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>magnetic_flux</td>
<td>length^2 * mass * charge^-1 * time^-1</td>
<td>webers</td>
<td>default_webers_domain</td>
</tr>
</tbody>
</table>

3.5 States, State Decompositions and Contexts

There is more to a datum than just its own domain. (Example.) Contexts and states exist to specify the time, time duration and other information on the applicability of a particular object's attribute's value.

3.5.1 States in Statements
States allow us to associate different values of one attribute with different times. They are grouped by how long they last. Attribute `states_finiteness_attr` maps from a state to a member of `finiteness_class` (with members `{infinitesimal, finite, infinite}`). Each is discussed below.

1. Infinitesimal states last just a negligible moment. As such, their starting, ending and midpoint times are all the same. They are used when Scilog applies mathematical methods that consider infinitesimally small changes, like integrating functions. By convention their only useful attributes of `states_finiteness_attr` and `states_time_attr` which maps from states to their midpoint times.

2. Finite states last for a finite (but perhaps unspecified) amount of time. They may follow one another. A finite state's previous and following state are may be queried with `states_prev_state_attr` and `states_next_state_attr` attributes respectively. Finite states may be subdivided along two axes: timed vs. untimed states, and series vs. non-series states.

3. An infinite state lasts forever. It covers all time: all of the past, the present, and all of the future. The pre-defined infinite state `universal_state` is the only one that is needed. The state `universal_state` is the default state of values and queries: if no state is explicitly given for a value of a query then its state will be `universal_state`.

Finite states, by definition, last a finite amount of time. By convention the time associated with a finite state is its middle. The attribute `states_midpoint_time_attr` maps from a state to this time. There are also the attributes `states_begin_time_attr`, `states_end_time_attr` and `states_durations_attr`. The following commonly expected relationships hold among those attributes:

\[
\text{states_durations_attr} = \text{states_end_time_attr} - \text{states_begin_time_attr}
\]

\[
\text{states_midpoint_time_attr} = \frac{0.5 \times (\text{states_begin_time_attr} + \text{states_end_time_attr})}{\text{states_end_time_attr} - \text{states_begin_time_attr}}
\]

The semantics of associating a finite state to a composite value is that the composite value's primary value is taken the mean value (for `float_pt`, `fixed_pt` values), median value (`integer`) or mode value (`concept`) over the duration of that state.

That definition gives us the following semantics. Say we are keeping track of the vertical position of an object as it falls with a time resolution of 0.25 seconds. Say we actually release the initially stationary object from position of 0.000 meters at time 0.00000 seconds according to some highly accurate stopwatch. Let there be a state `dropping_object_state0` with midpoint time 0.000 seconds (during with the object was dropped) actually extents from -0.125 seconds (begin time) to +0.125 seconds (end time). The dropping event began exactly at that state's midpoint time (within precision limitations). This is what we agrees with our intuition.

However, because the `dropping_object_state0` extends from -0.125 seconds
to +0.125 seconds there is a non-intuitive result. The downward velocity was 0.00 meters/second for the first half of the state (-0.125 to 0.000 seconds), but it accelerated for the second half of the state (0.000 to +0.125 seconds). This means that its average velocity during this state is non-zero.

3.6 Comparisons and Operations

3.7 Input/Output
4 Static Knowledge: Equations and Decision Trees

Static knowledge is knowledge that is considered (relatively) unchanging and timeless. Examples include the boiling points and melting points of materials. The Scilog represents this knowledge with equations and decision trees that are not associated with processes. Equations and decision trees are two common types of knowledge-interrelating clauses called assertions.

4.1 Static Equation Basics

Equations relate numerical values among multiple entity attributes. Ideally if all but one such numerical value is known then the last one may be uniquely computed. For example, let us return to the density equation 1-1, but be more detailed. Our result is given in 4-1.

Equation 4-1: The density equation, revisited.

\[ \forall (x \in \text{physical_object_class}) \colon \text{uniform_density}(x) \rightarrow \text{mass}(x) = \text{volume}(x) \times \text{density}(x) \]

which states that the mass of any physical object \( x \) with uniform density is equal to the product of its volume and its density. It is represented by the static equation \( \text{density}_\text{eqn} \) which can be shown by following Edit Knowledge Component -> Theory, and then by choosing \( \text{density}_\text{eqn} \) under Equation(s). This displays the window below:

Figure 4-1: The equation \( \text{density}_\text{eqn} \):

All equations are 4-tuples of predicate \textit{equation}. The first argument is a unique name for the equation. In this case it is \( \text{density}_\text{eqn} \).

The second argument is a list of entity variables and their classes. The format for each
entry in this comma-separated list is:

**Figure 4-2: The High Level Language format for specifying entity variables and the classes that are their domains.**

```
<variable_name> instance_of <class_name>
```

Here, `<variable_name>` represents a variable over domain entities, **not** attributes. In equation 4-1 it is represented by variable `x`. In `density_eqn` it is represented by symbol `object`. The symbol may be any legal Prolog user-defined symbol. Ideally symbols used as entity variables do **not** appear in Scilog's ontology, otherwise it is not clear if it is being used as an variable or a constant representing a concept.

The occurrence of keyword `instance_of` is required.

Lastly, `<class_name>` gives the name of the class that entity variable `<variable_name>` may range over. It must be class in the Scilog ontology. In `density_eqn` we have “object instance_of physical_object_class” meaning that the entity variable `object` may range over any instance in class `physical_object_class`. This corresponds to the “∀(x ∈ physical_object_class)” of equation 4-1.

The third argument is a list of conditions that must hold for the equation to be applicable. There are several formats for conditions, and they are covered below. Perhaps the most common is:

**Figure 4-3: The High Level Language formats for specifying assertion conditions.**

```
(<entity>.<attribute> == <value>)
(<entity0> instance_of <entity1>)
(<entity0>.<attribute0> > <entity1>.<attribute0>)
(<entity0>.<attribute0> >= <entity1>.<attribute0>)
```

In the first form `<entity>` may either be an entity variable introduced in the previous list or an object in the ontology. `<attribute>` may either be a fixed reference to an attribute of that entity or an entity variable that specifies the attribute dynamically. High-level language Scilog (that is entered in GUI windows) follows the C/C++/Java convention of referring to an attribute of an object by having a period (".";) between the object `<object>` and its attribute `<attribute>`.

We then have the Prolog comparator `==` meaning “the things on either side must be each other”. Other comparators include `>`, `>=`, `<` and `<=`, all of which have their conventional meanings. (Prolog’s “less than or equal to” comparator is `<=` which seems unique to Prolog. In Scilog recognizes both `<=` and `=<`, and both mean the same thing. `=<` is used by High Level Language to display the comparator “less than or equal to.”)

Finally, there is `<value>`. This may be an actual constant value, another entity variable, or another `<entity>.<attribute>` pair.
In the second form the requirement is that `<entity0>` belong to the set specified by `<entity1>`. This is in addition to `<entity0>` having to belong to the class given in the entity class list.

The third and fourth condition forms are similar to the first. They state that `<entity0>`'s value for `<attribute0>` must be greater than (or greater than or equal to) `<entity1>`'s value for `<attribute1>`.

The sole condition of `density_eqn` is that object's value for `has_uniform_density_attr` must be equal to `true`.

The fourth argument to an equation predicate is the equation itself in traditional infix format. The equation must have two sides that are related to each other by the comparator `:=`. Both sides may use constants and `<entity>.<attribute>` value references that are combinable with arithmetic and many common unary operators. (See below.)

In `density_eqn` we have stated that object's value for `mass_attr` is equal to its value for `volume_attr` times its value for `objects_material_density_attr`.

Scilog tries to apply equation knowledge very generally. It is clever enough to solve `density_eqn` for either `volume_attr` or `objects_material_density_attr` if required to further a calculation.

Although Scilog will try to solve equations as needed, the form that the equation is given is in is very important. If the equation is given in form:

**Figure 4-4: The High Level Language format for specifying equations that define an entity and its attribute.**

```plaintext
<entity>.<attribute> := some_function()
```

then `<entity>` becomes the “defined entity” and `<attribute>` becomes the “defined attribute” of the equation. For example, Scilog's default behavior is to try to arrange equations from most specific to most general. It does this by looking at how broad the ontology class for entity `<entity>` is. Also, when Scilog tries to compute which attribute values are changed when the knowledge base is changed it looks at the defined `<entity>` and `<attribute>` values of equations and decision trees. If by changing the knowledge base some value in the `some_function()` portion of the equation may change then pairing `<entity>.<attribute>` is also said to change. This could recursively cause other equations and decision trees to change their predictions. This “closure of changes” is needed to compute a fuller range of changes implied by a change to the knowledge base.

4.2 Static Decision Tree Basics
Decision trees use the symbolic knowledge of several attributes to compute the value for one particular attribute. For example, the decision tree below predicts the value of attribute objects_material_density_attr for some instance of class physical_object_class.

Figure 4-5: Decision tree to predict the density of an object given the material and physical phase of an object.

```
objects_material_attr =
    lead
    latex

objects_material_phase_attr =
    common_solid

11.35 gm/ml
0.92 gm/ml
0.5 gm/ml
1.00 gm/ml
```

The decision tree above shows that to compute a value for an object's objects_material_density_attr attribute, that object's value for objects_material_attr should first be computed. If it is lead, and the object's value for objects_material_phase_attr is common_solid, then the computed density is 11.35 gm/ml. Similarly, if the object's value for objects_material_attr is latex and its value for objects_material_phase_attr is common_solid, then the computed density is 0.5 gm/ml. Finally, if objects_material_attr is h2O then the value for objects_material_phase_attr should be computed. If it is common_solid then the density is 0.92 gm/ml but if it is common_liquid then it is 1.00 gm/ml.

The decision tree material_density_tree may be shown by going to Edit Knowledge Component and then choosing Law. Then, under D-Tree(s) choose material_density_dtree. The window below should appear.

Figure 4-6: Decision tree material_density_dtree.
All decision trees are 5-tuples of predicate dtree. The first argument is a unique name for the decision tree. In this case it is material_density_dtree.

The second argument is an entity variable that represents the thing object for which an attribute is computed. More will be said about it when discussing all entity variables in the fourth argument.

The third argument is an attribute of the previous entity that is being computed. In the example above it is objects_material_density_attr.

The fourth argument is a comma-separated list as given in figure 4-2. This list is identical in format and serves the same purpose as the second argument to static equation clauses. (Please see 4.1.) A requirement unique to decision trees is that this list must define the entity variable specified as the second argument to the dtree clause as one of the <variable_name>'s. It is strongly recommended that the entity variable given in the second argument to the dtree clause be the first entity variable defined in this list.

The fifth and last argument is a list that should be left empty in High Level Language. The GUI will place the definition of the tree in this area. Scientists may inspect the tree by clicking on the various nodes of the left hand side panel. Scientists may also edit the tree with the New Node, Copy Node, Paste Node and Delete Node buttons given on the bottom of the right hand side panel.

Figure 4-4: Decision tree material_density_dtree (tree expanded).
The tree allows three broad types of nodes. Attribute testing nodes specify an entity and its attribute to compute. In `material_density_dtree` all nodes labeled `object.objects_material_attr` and `object.objects_material_phase_attr` are attribute testing nodes. Underneath attribute testing nodes are one or more matching nodes. Matching nodes specify a value to match the attribute test node’s resulting computed value against, and a test (generally equality) with which to compare the two values. In the nodes `=:= (common_solid)` and `=:= (common_liquid)` are matching nodes. Underneath the matching nodes are either another attribute testing node or a value node which finally specifies the predicted value for the decision tree. In the example above all value nodes specify a density, like `value(0.92, default_grams_per_ml_domain).

Decision trees always have an explicit defined `<entity>` and `<attribute>` (the second and third arguments respectively) which serve the same purpose as the defined `<entity>` and `<attribute>` of equations.

Decision trees cannot, however, be used as flexibly as equations. Equations (potentially) can be solved for entities and attributes other than the defined entity and attribute. Scilog does not do this for decision trees.

4.3 Advanced Static Equations and Decision Trees

4.3.1. Overview

Static equations and decision trees are represented stylized Prolog ground clauses with the predicates `equation` and `dtree` respectively. Please note the following:

1. The `equation` and `dtree` predicates must be a ground clauses. These clause must have no Prolog variables.
2. The equation predicate is a 4-tuple.
   a. The first argument is the equation's unique name.
   b. The second argument is a comma-separated list of classes for each object entity that the equation interrelates. The ordering of this list is important: throughout the rest of the clause the first listed class corresponds to the entity identified with integer 0, the second listed class corresponds to the entity identified with integer 1, etc.

   The mapping between the entity index integers that Scilog natively uses and the more mnemonic entity variables is specified with the highlevel_entity_name_map predicate. Clauses that use this predicate have two arguments. The first is the name of an assertion that they describe (e.g. density_eqn or material_density_dtree). The second is an ordered list of variable names. Entity integer 0 will be represented in High Level Language by the first name in the list, entity integer 1 will be represented in High Level Language by the second name in the list, etc.
   c. The third argument is a list of conditions among objects that must be satisfied before the equation is applicable. Conditions are discussed below.
   d. The fourth argument is a parse tree of the equation. The parse trees are discussed below.

3. The dtree predicate is a 6-tuple.
   a. The first argument is the decision tree's unique name.
   b. The second argument is an integer representing the defined entity for which this decision tree predicts. (See the discussion of the fourth argument below.)
   c. The third argument is the defined attribute of the defined entity that this decision tree predicts.
   d. The fourth argument is a comma-separated list of classes for each object entity that the decision tree interrelates. The ordering of this list is important: throughout the rest of the clause the first listed class corresponds to the entity identified with integer 0, the second listed class corresponds to the entity identified with integer 1, etc.

   The mapping between the entity index integers that Scilog natively uses and the more mnemonic entity variables is specified with the highlevel_entity_name_map predicate. Clauses that use this predicate have two arguments. The first is the name of an assertion that they describe (e.g. density_eqn or material_density_dtree). The second is an ordered list of variable names. Entity integer 0 will be represented in High Level Language by the first name in the list, entity integer 1 will be represented in High Level Language by the second name in the list, etc.
   e. The fifth argument is a list of conditions among objects that must be satisfied before the decision tree is applicable. Conditions are discussed below.
   f. The sixth argument is the decision tree. The parse trees are discussed below.

4.3.2. Conditions
Equation and decision tree condition lists tells when the assertion is valid. It is a conjunction of zero or more individual conditions, all of which must be met for the equation to be valid.

There are seven types of conditions:

1. value-cond(ENT_NUM,ATTR,VAL): The entity numbered ENT_NUM must have an attribute ATTR with value VAL. Here ENT_NUM is an entity index integer and VAL is a value. ATTR must either be an attribute or the index integer of an entity that corresponds to some member of attribute_class.
2. frame-cond(OBJ,ATTR,VAL): Object OBJ must have an attribute ATTR with value VAL. Here OBJ must be an object in the ontology and VAL is a value. ATTR must either be an attribute or the index integer of an entity that corresponds to some member of attribute_class.

Both value-cond and frame-cond map to the High Level Language form (<object>.<attribute> == <value>) where the only difference is whether or not <object> is an object or entity variable. (This distinction is syntactically irrelevant to High Level Language programming.)

3. entity_value-cond(ENT_NUM0,ATTR,ENT_NUM1): The entity numbered ENT_NUM0 must have an attribute ATTR with value being the concept bound to entity ENT_NUM1. Here ENT_NUM0 and ENT_NUM1 are entity index integers and ATTR is an attribute. This maps to the High Level Language form (<object0>.<attribute> == <object1>).
4. entity_instance-cond(ENT_NUM0,ENT_NUM1): The entity numbered ENT_NUM0 must belong to class given by the entity numbered ENT_NUM1. This is in addition to belonging to the class stated in the entity class list. Both ENT_NUM0 and ENT_NUM1 are entity index integers where the latter one represents a class. This maps to the High Level Language form (<object0> instance_of <object1>).
5. entity_value_equality-cond(ENT_NUM0,ATTR0,ENT_NUM1,ATTR1): The entity numbered ENT_NUM0 must have an attribute ATTR0 with value that equals ENT_NUM1’s value for attribute ATTR1. Here both ENT_NUM0 and ENT_NUM1 are entity index integers and ATTR0 and ATTR1 are their attributes respectively. Either or both ATTR0 and ATTR1 may be entity variables that correspond to attributes. This maps to the High Level Language form (<object0>.<attribute0> == <object1>.<attribute1>).
6. entity_value_greater_than-cond(ENT_NUM0,ATTR0,ENT_NUM1,ATTR1): The entity numbered ENT_NUM0 must have an attribute ATTR0 with value that is greater than ENT_NUM1’s value for attribute ATTR1. Here both ENT_NUM0 and ENT_NUM1 are entity index integers and ATTR0 and ATTR1 are their attributes respectively. Either or both ATTR0 and ATTR1 may be entity variables that correspond to attributes. This maps to the High Level Language form (<object0>.<attribute0> > <object1>.<attribute1>).
7. entity_value_greater_equal-cond(ENT_NUM0,ATTR0,ENT_NUM1,ATTR1): The entity numbered ENT_NUM0 must have an attribute ATTR0 with value that is greater than or equal to ENT_NUM1’s value for attribute ATTR1. Here both ENT_NUM0 and ENT_NUM1 are entity index integers and ATTR0 and ATTR1 are their
attributes respectively. Either or both \texttt{ATTR0} and \texttt{ATTR1} may be entity variables that correspond to attributes. This maps to the High Level Language form \((\texttt{object0}.\texttt{attribute0}) \geq \texttt{object1}.\texttt{attribute1})

**Equation parse tree**

The equation parse tree states the given numeric relation. It is constructed from operator nodes and leaves that reference specific values.

Operator nodes state which numeric operation should be applied to the parameters (subtrees) of the node. The highest-level node for an equation must be the binary numeric equality procedure \(=\) and this operator may only appear at this location. The subtrees of this highest-level node may be operators or references to specific values.

**Operators**

1. **Nullary operators:** Nullary operators have no operands and no parentheses. Supported nullary operator(s) include:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand_0_1</td>
<td>Returns a random number between 0 and 1.</td>
</tr>
</tbody>
</table>

   **Nullary Scilog numeric operators**

   2. **Unary operators:** Unary numeric operators have one subtree operand. Supported unary operators include:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(X)</td>
<td>Returns the absolute value of X. The result gets the same datatype as X has.</td>
</tr>
<tr>
<td>acos(X)</td>
<td>Returns the arccosine of X. The result is a number with dimensions \texttt{angle} and units \texttt{radians}.</td>
</tr>
<tr>
<td>asin(X)</td>
<td>Returns the arcsine of X. The result is a number with dimensions \texttt{angle} and units \texttt{radians}.</td>
</tr>
<tr>
<td>atan(X)</td>
<td>Returns the arctangent of X. The result is a number with dimensions \texttt{angle} and units \texttt{radians}.</td>
</tr>
<tr>
<td>cos(X)</td>
<td>Returns the cosine of X. X must be or must be bound to a number with dimensions \texttt{angle} and must either have units \texttt{degrees} or \texttt{radians}. The return value has dimensions \texttt{dimensionless} and units \texttt{unitless}.</td>
</tr>
<tr>
<td>exp(X)</td>
<td>Returns e raised to the power of X. The result has dimensions \texttt{dimensionless} and units \texttt{unitless}.</td>
</tr>
</tbody>
</table>
exp10(X) Returns 10 raised to the power of X. The result has dimensions *dimensionless* and units *unitless*.

log(X) Returns the natural logarithm of X. The result has dimensions *dimensionless* and units *unitless*.

log10(X) Returns the logarithm-base-10 of X. The result has dimensions *dimensionless* and units *unitless*.

max(X,Y) Computes X and Y and returns the maximum value between the two.

min(X,Y) Computes X and Y and returns the minimum value between the two.

sin(X) Returns the sine of X. X must be or must be bound to a number with dimensions *angle* and must either have units *degrees* or *radians*. The return value has dimensions *dimensionless* and units *unitless*.

tan(X) Returns the tangent of X. X must be or must be bound to a number with dimensions *angle* and must either have units *degrees* or *radians*. The return value has dimensions *dimensionless* and units *unitless*.

2. **Binary operators**: Binary numeric operators have two subtree operands. Supported binary operators include:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>+(X,Y)</td>
<td>Returns the sum of X and Y. X and Y must share the same dimension and must be in convertable units.</td>
</tr>
<tr>
<td>-(X,Y)</td>
<td>Returns the difference of X minus Y. X and Y must share the same dimension and must be in convertable units.</td>
</tr>
<tr>
<td>*(X,Y)</td>
<td>Returns the product of X and Y.</td>
</tr>
<tr>
<td>/(X,Y)</td>
<td>Returns the quotient of X divided by Y.</td>
</tr>
<tr>
<td>pow(X,Y)</td>
<td>Returns X raised to the power of Y.</td>
</tr>
</tbody>
</table>

2. **Trinary operators**: The only supported trinary operator are decision tree nodes, which may appear in equations. The syntax is the same as it is in decision tree sentences:

: **Syntax of get_value(,) procedure**
where `entity_index` is the entity index of the object in question, and `attribute` is its attribute, and `choice_list` is a list of `dtree_choice` terms as detailed in 5.3.2. An advantage of having decision tree nodes in equations is conciseness. For example, consider the intensity of light as it goes through a vacuum, then through one opaque material that partially absorbs it, and then through another. The intensity as a function of distance traveled can be given as:

(EQ 2)
5 Lists, Decompositions, Finite State Machines and Context Free Grammars

5.1 Lists

Lists can be used to efficiently store measurements as well as to tell the parts of objects. Lists are such a common datastructure that there is a special predicate for it. Its simpler form is:

```
list(<list_name>, [<first_obj>, <second_obj>, ... <last_obj>]).
```

The main internal usage of lists is to define the decompositions of things, specifically states, process instances and arbitrary objects. This is done with the attributes `objects_decomposition_attr` which maps from arbitrary objects to a decomposition object, and `decompositions_list_attr` which maps from a decomposition object to the name of a list. The named list represents a complete ordered set of non-overlapping components into which the given object can be decomposed.

### Attributes of Lists

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lists_length_attr</td>
<td>Maps from a list to the number of items that it has. It is automatically computed.</td>
</tr>
</tbody>
</table>

By default the objects in the list have no knowledge that they are in a list, and neither does any other object. List information can be associated with the objects by an optional third parameter to the `list` predicate.

In direct augmentation list information is directly associated with each item in the list. This is appropriate if there are only one (or maybe two) decompositions that each item in the list can be in. SciLog does this by specifying attribute names. If the term `next_item_attr_attr(<next_attr>)` is present it defines a slot for each item in the list that denotes that its value for attribute `<next_attr>` is the next item in the list (or is undefined for the last list item). If the term `prev_item_attr_attr(<prev_attr>)` is present it defines a slot for each item in the list that denotes that its value for attribute `<prev_attr>` is the previous item in the list (or is undefined for the first list item). If the term `listed(<list_attr>)` is present it defines a slot for each item in the list that denotes that its value for attribute `<list_attr>` is the list that it is in.

In indirect augmentation list information is associated with invented objects each item in the list. This is appropriate if there are potentially many decompositions that an item in the list can be in.

5.1.1. Decompositions of states

The items listed in a finite decompositions of states must themselves be the substates
of the given state. They must be non-overlapping and all inclusive.

5.1.2. Decompositions of process instances

The items listed in a finite decompositions of process instances must themselves be the subprocess instances of the given process instance. They must be non-overlapping and all inclusive.

5.2 Iterators over lists and finite decompositions

To iterate over the items in a list or finite decomposition tell the context which thing you are interested in. This is done by setting the attribute contexts_list_attr in a context to the list (or finite decomposition). When this attribute is defined (or re-defined) a Scienceomatic iterator object is created that supports iterator functionality. Access the context's iterator's functionality using the context attributes in the table below.

Note, this is also defined to iterate finite decompositions, whether or not they are implemented with decompositions_member_class_attr or decompositions_list_attr (see 5.3 Decompositions).

<table>
<thead>
<tr>
<th>Context Attributes for Iterating Over Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>contexts_list_attr: Maps from context to current list being iterated over. When this is set it has the side effect of setting context_lists_index_attr to 0 in the same context.</td>
</tr>
<tr>
<td>context_lists_index_attr: Maps from context to an integer greater than or equal to 0 that defines position in list. The first object has index 0. If this is set (and contexts_list_attr is defined in the same context) then the next value returned for context_lists_item_attr and context_lists_next_item_attr is changed accordingly. Thus, lists may be treated as read-only arrays.</td>
</tr>
<tr>
<td>context_lists_item_attr: Maps from context to the current item of the context's list. Does not advance iterator so multiple calls will return the same item.</td>
</tr>
<tr>
<td>context_lists_next_item_attr: Maps from context to the next item of the context's list. Does advance iterator so multiple calls will the next item. Increments context_lists_index_attr.</td>
</tr>
</tbody>
</table>

Rewinding the iterator back to its first item may either be accomplished by setting contexts_list_attr to its old value, or by setting context_lists_index_attr to 0.

5.3 Decompositions

Decompositions map from objects to ways that they may be divvied up into complete
and non-overlapping component items sets. All decompositions should be members of decomposition_class.

Users may tell how to decompose objects with the multi-valued attribute objects_decomposition_attr which maps from objects to their decompositions. One unique decompositions may be specified for each object with the single-valued attribute objects_canonical_decomposition_attr. The multi-valued, dynamically-constructed attribute objects_composition_attr maps from an object to a finite decomposition that gives that object as an included item.

There are three types of decompositions: (1) finite, class-membership specified; (2) finite, list-specified; and (3) infinitesimal. These three types are distinguishable by the presence or absence of decomposition attributes decompositions_member_class_attr and decompositions_list_attr, and by the value of attribute decompositions_finiteness_attr (either finite or infinitesimal).

Finite, class-membership specified decompositions have a finite number of parts which are all members of a given class. The attribute decompositions_member_class_attr maps from the decomposition to the class.

Finite, list-specified decompositions have a finite number of parts which are given in a linked list. The attribute decompositions_list_attr maps from the decomposition to the list.

Infinitesimal decompositions have an infinite number of parts and are used for integration. The attribute decompositions_independent_attr maps from the decomposition to the attribute that is the independent variable of integration. The attribute decompositions_independent_lo_val_attr maps from the decomposition to the independent variable's lowest value. The attribute decompositions_independent_hi_val_attr maps from the decomposition to the independent variable's highest value.

decomposition_class (is_a cultural_convention_object_class)

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decompositions_object_attr</td>
<td>Maps to an object whose decomposition is described. The inverse attributes objects_decomposition_attr (multi-valued) and objects_canonical_decomposition_attr (single-valued) map from objects to members of decomposition_class.</td>
</tr>
<tr>
<td>decompositions_finiteness_attr</td>
<td>Maps to either finite if there are a finite number of parts or infinitesimal if there are an infinite number. Additive attributes of decomposed objects may be computed from their component's values of that attribute by summation for finite decompositions and by integration for infinitesimal decompositions.</td>
</tr>
<tr>
<td>decompositions_member_class_attr</td>
<td>(for finite decompositions) Maps to class of which all components are members.</td>
</tr>
<tr>
<td>decompositions_list_attr</td>
<td>(for finite decompositions) Maps to list that enumerates parts of decomposition. The values in the list must all be concepts. Further, these each value must occur no more than once in the list.</td>
</tr>
<tr>
<td><strong>decomposition_class</strong> <em>(is_a cultural_convention_object_class)</em></td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td></td>
</tr>
</tbody>
</table>
| **decompositions_independent_attr_attr**: (for decompositions described by decompositions_object_attr) Maps from infinitesimal decomposition to attribute that is its independent integration variable. It's the "x" in  
\[
\int_a^b f(x) \, dx
\]
| **decompositions_independent_lo_val_attr**: (for decompositions described by decompositions_object_attr) Maps from infinitesimal decomposition to the lower value that the decomposition's decompositions_independent_attr_attr should be in integration. It's the "a" in  
\[
\int_a^b f(x) \, dx
\]
| **decompositions_independent_hi_val_attr**: (for decompositions described by decompositions_object_attr) Maps from infinitesimal decomposition to the upper value that the decomposition's decompositions_independent_attr_attr should be in integration. It's the "b" in:  
\[
\int_a^b f(x) \, dx
\]
| **is_decomposition_equally_timed_attr**: Maps to true if decomposition of object (state, process, etc.) has parts that all have the same duration and occur sequentially, or to false otherwise.  
**is_decomposition_temporally_parallel_attr**: Maps to true if decomposition of object (state, process, etc.) has parts that all last throughout the whole object's duration, or to false otherwise.  
**compositions_immediate_subcomposition_attr**: (A derived attribute)  
**compositions_subcomposition_attr**: (A derived attribute) |
6 Dynamic Knowledge: Process Classes and Instances

Processes describe how a system changes from a before state to an after state over time. Processes may be decomposed into subprocesses to describe finer-grained state changes.

Processes that are mathematically smooth also allow us to compute what happens in substates during the state given. For example, they let us describe the changes that happen when objects like lead_weight are released.

Like Langley et al [2002] we distinguish between abstract and concrete processes. We call abstract processes process classes, and we call concrete processes process instances.

6.1 Process classes

Process classes are meant to represent a set of possibly similar phenomena. They have:

1. a class which name uniquely identifies it and serves as the name of the set of process instances that are examples of it,
2. an object list which gives the classes of objects that are interrelated by the process’ effects. For example, the process dropping_object_process_class defines the position, velocity and acceleration for members of dropping_object_class.
3. a conditions list which state a set of predicates that must be true for instances of this process class to be in effect, and,
4. a manifestations list which give relationships that pertain to all instances of this class. Both equations and decision trees may be in the manifestations list but they use the special terms proc_equation and proc_dtree. They are both similar to ordinary equations and decision trees but they use the entity list and condition list of the process class to which they belong.

The proc_equation term has two parameters: the first is a name that uniquely identifies the equation and the second is the equation’s parse tree (please see 6.3).

The proc_dtree term has four parameters. The first is a name that uniquely identifies the decision tree. The second and third are the entity and attribute respectively of the process that are being computed. The last is the decision tree itself (please see 5.3.2).
6.2 Basic process instance computation

Process instances concern one particular happening of a process class. Instances have a name, a process class to which they belong, a state, and name of all objects that that process instance interrelates. An example process instance is given in the figure below.

: Process instance example

This process instance defines the motion of lead_weight.

This process instance, when combined with some initial information and some facts about the objects, can be used to compute attributes of inter-related objects. There are two types of direct computation:

1. using the preconditions, and
2. using the manifestations

Computing with manifestations just uses the given equations and decision trees. Also, because the semantics of process instances is that they record actual events, we know their preconditions must have been met. Therefore, preconditions may used as sources of knowledge.

: Example of basic computation with process instances:

Sometimes a process instance should tell how all objects in an entity class change instead of some particular instance of a class. In these cases the term any_member in the list of related objects means that process instance describes how all members of the object’s corresponding class behave. For example, :

: Example process class:

This process instance should describe the translation of all points in lead_weight and therefore should be any_member:

: Example usage of any_member process instance:

6.3 Composite process instance computation

Like Langley et al [2002] our process instances may be composed of other processes. The two ways that a process instance may be decomposed are temporally (implemented by serial states) and structurally (implemented by identifying conceptually simpler subprocesses that act in parallel).

The attribute processes_subprocess_division_attr maps from processes to an object that tells how that process can be decompose. The two attributes subprocess_divisions_serial_process_attr and subprocess_divisions_parallel_process_attr map from the process
decomposition object to process instances that are either active in serial (all at the same time) or in parallel (one after another).

**Example definition for composite process instances**

Composite relationships can be used to constrain both the “super” process instance and the “sub” process instance. Additionally, the attribute `processes_subprocess_attr` may be recursively queried.

**Example of composite computation with process instances:**

In `red_ball_motion_towards_wall` the `red_ball` spends 1 second moving in the positive x direction while in `red_ball_motion_away_from_wall` the `red_ball` spends 1 second moving in the negative x direction. Scilog takes the weighted sum of these to compute the x momentum value 0.

Sometimes process instance knowledge is redundantly stored at different states and with different subprocesses. This serves two purposes:

1. It aids efficient computation by caching results at states and subprocesses the `scientists` state as being useful (cf the utility problem).
2. It allows for composite processes by combining the effects of different small state processes into one large state process.

### 6.4 Object-oriented processes instance computation

Process classes exist in a hierarchy so process instances may draw upon the knowledge of ancestreral process classes. However, before this can be done, you must tell Scilog how the entities of the subprocess class relate to the entities superprocess class. This is done with the `process_class_inherit` predicate. It has the following syntax:

**Syntax of the `process_class_inherit` procedure:**

```
process_class_inherit(<subprocess class>,
                     <superprocess class>,
                     <entity map list>)
```

The argument's meanings are described below:

1. `<subprocess class>` is the subprocess class that is being related to a superprocess.
2. `<superprocess class>` is the superprocess class that is being related to a subprocess.
3. `<entity map list>` is a list of `entity_map` functions. Each function either map one subprocess class entity to a superprocess class entity, or map an object to a superprocess class entity. Every superprocess class entity must be mentioned precisely one time.
while several subprocess entities may map to the same superprocess class.

For example, 

: Derived process classes:

The figure above defines a new process class __ that derives from __. __. Notice that the list skips some subprocess entities but defines all superprocess entities.
We may now use these parent-child pairing to compute some properties defined in __. For example, __:

:  

6.5 Processes instance computation using calculus

If Scilog is told how one attribute may be obtained from another through integration then it can use calculus to compute values. The syntax for stating these relationships is:

: Syntax of the integratable_attribute_relation procedure:

integartable_attribute_relation(<object class>,
   <resultant attr>,
   <dependent attr>,
   <independent attr>,
).

where:
1. <object class> is the class of object for which attributes of this relationship hold.
2. <resultant attr> is the attribute that results from integration.
3. <dependent attr> is the attribute that is integrated to yield <resultant attr>.
4. <independent attr> is the attribute that of integration. Some function of <dependent attr> is integrated with respect to <independent attr> to yield <resultant attr>.

For example, for physical objects if one integrates some function of acceleration with respect to time one gets velocity.

: Velocity results from integrating acceleration as a function of time:

integartable_attribute_relation(physical_object_class,
   velocity_attr,
   acceleration_attr,
   states_midpoint_time_attr
).

The following is a simple application. Process equation __:

: Process class
We also have to tell it the relationship the relationships among the instantaneous states just before, during and just after the experiment:

: State relationships

Finally, we have to tell it when the instantaneous states were in time (at least, relative to each other):

: State times

Using all this information we can ask Scilog __.

: Query:

<table>
<thead>
<tr>
<th>Attributes of Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_process_mathematically_smooth_attr: Maps from a process to either true or false. If it maps to true that means that for all changes included in that process and for all equations grouped with that process, all derivatives of all possible solved forms of the equations have only smooth derivatives. This lets Scilog apply equations for substates of the given state of a process instance. For example, dropping_object_process_class is mathematically smooth. Thus we can apply its equations even for subclasses of the given state of a process instance.</td>
</tr>
</tbody>
</table>
7 Simulations
8 Abduction and Induction
9 Architecture and Reasoning

6.1 Architecture

The Scientific Metareasoner is implemented by five Scilog component objects that may be placed into two groups: **empirical** (theory, law and data) and **non-empirical** (metaphysics and analytics). The empirical components hold science: scientific data and scientific generalizations. The non-empirical components hold cultural background knowledge, how things relate to each other (ontology) and how to recast problems from one form to another (e.g. algebra and calculus).

This subchapter will discuss all five components.

1. **metaphysics**. This component holds the framework in which all other knowledge is organized and also some standard Prolog predicates. This includes:
   (a) the ontology (the relationships among concepts),
   (b) some standard Prolog predicates,
   (c) assignment of properties to non-empirical entities, and,
   (d) limitations on the empirical model.

   The ontology defines the relationship among all concepts known by the reasoner in an is-a hierarchy. It is given in the files with extension .ontol.pro, which must have all and only is_a(subclass,superclass) and instance_of(instance,class) predicates. In the figure above the file std.ontol.pro gives the definitions of standard domain-independent Scilog objects (e.g. standard non-empirical attributes) while the file tutorial.ontol.pro domain-specific objects.

   Some of the standard prolog predicates that are defined in the metaphysics include member(item,list) and conc(sublist0,sublist1,combinedList). These are given in std.back.pro.

   Much of the metaphysical knowledge assigns properties to things. The file std.back.pro assigns arity, associativity and other properties to numeric operators; it defines composite dimensions (e.g. area, force) in terms of atomic ones (e.g. length, time, mass); it associates units (e.g. meters) with the dimensions that they measure (e.g. length); it tells how some units (e.g. cm) relate to others (e.g. 100 cm = 1.0 meters); it assigns datatypes, units, precision information and upper and lower bounds to domains (e.g. default_degrees_neg_180_to_180_domain is measured in degrees, has an upper bound of 180, a lower bound of -180), it also gives miscellaneous information (e.g. telling if pre-defined attributes are single-valued or not). The file tutorial.back.pro does much the same thing for domain-specific attributes and domains.

   Finally, the metaphysical Scilog files give limitations on what the rest of the model may assert. This may be stated in terms of physical properties, equations, decision trees and process classes that a scientist takes as truly fundamental and not subject to
reconsideration (e.g. “God-given”). It may also be stated with the (not yet supported)
new constraints on model (“tier 1”) predicate
frame_for_study(class, attribute, component_list) and new constraints on constraints (“tier 2”) predicate
disallow_predicate(predicate_form).

frame_for_study signifies that the attribute attribute of members of class
class may only be specified/used/elaborated in the components mentioned in
component_list. A special case is when component_list is empty or just specifies
metaphysics. For example, a religious person might want to say that “God's will”
is just a given and is not open to further consideration:

frame_for_study(deity_class, being_wants_attr, [metaphysics]).

disallow_predicate specifies that predicates of form predicate_form
are not allowed. For example, an agnostic or atheistic scientist may want to state that
everything is open to further study.

disallow_predicate(frame_for_study(_, _, [metaphysics])).

This says “disallow any attempt to specify the study of anything down to just
metaphysics”.

The (not yet supported) command
is_there_metaphysical_inconsistency will check tier 1 constraints with
the rest of the model and will check tier 2 constraints with the rest of metaphysics.

2. theory. This component gives high level (“theoretical”) domain knowledge. It is
meant for the most general and highlevel knowledge that is applicable to a variety of
cases, but to no particular case in without more detailed knowledge. We chose to put
the equation:

mass = density * volume

in this component because it is broadly applicable to any instance of
physical_object_class, but needs at least two specific values for any object to
be immediately useful.

3. law. This component gives intermediate level domain knowledge. It is meant for
knowledge that bridges the gap between theory and data, for generalizations of data
that do not rise all the way to theory because theory is more general still. Assertions
that deal with more specific cases in greater detail are one example of the type of
knowledge intended for this component. Another example is the decision tree that
returns the density of an object based upon its material and material phase. These data
are generalizations over many measurements, but pertain to a very specific aspect
(only one attribute) of objects.
4. **data.** This component gives low level data. It is meant to give ground facts corresponding to measurements about objects. We placed specific facts about specific objects in this component.

5. **analytics.** This component tells how to recast problems. The file `std.math.pro` gives an equation that defines random numbers for all objects and gives several equations that inter-relate a state's beginning time, ending time and duration. It also tells what attribute is obtained when another attribute is integrated (e.g. velocity results from integrating acceleration with respect to time). The file `calculus.pro` has the rules to do differential an integral calculus so this knowledge can be applied.

Only the analytics component is allowed to do calculus.

6.2 Reasoning

Reasoning is done by giving a query to the components in a defined order. If the first component cannot answer the query then the second is tried, and then the third, etc., until all either one of them returns an answer or the query fails. The component metaphysics is always checked first because it holds knowledge that is defined to be true. Likewise, analytics is always checked last because only after all other components have been checked should we try to recast a problem into another form. Recursive subqueries cause the system to check the components in the same order (with forced_theorize being an exception).

The defined orders of the computational approaches are given below:

<table>
<thead>
<tr>
<th>given</th>
<th>ab_initio</th>
<th>read_data</th>
<th>theorize</th>
<th>empiricize</th>
<th>forced_theorize</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>metaphysics</td>
<td>metaphysics</td>
<td>metaphysics</td>
<td>metaphysics</td>
<td>metaphysics</td>
</tr>
<tr>
<td></td>
<td>analytics</td>
<td>data</td>
<td>theory</td>
<td>data</td>
<td>theory</td>
</tr>
<tr>
<td></td>
<td>theory</td>
<td>analytics</td>
<td>law</td>
<td>analytics</td>
<td>law</td>
</tr>
<tr>
<td></td>
<td>analytics</td>
<td></td>
<td>data</td>
<td></td>
<td>analytics</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. The given computational approach consults only non-empirical components. Therefore it can be used to give non-scientific (but still reasoned) answers.
2. The ab_initio computational approach uses the non-empirical components and theory as its sole empirical component. It can be used to answer queries using first principles.
3. The read_data computational approach uses the non-empirical components and data as its sole empirical component. It answers queries by using the deductive closure of the data.
4. The theorize computational approach uses all of the components. It queries the more theoretical empirical components first and therefore returns a more theory-

---

2 forced_theorize will not use the data component to answer the primary given query but will use data as theorize does to answer any recursive subqueries.
laden answer.

5. The empiricize computational approach also uses all of the components. It queries the less theoretical empirical components first and therefore returns a more data-laden answer.

6. The forced_theorize computational approach is like theorize. It uses all of the components in the same order as theorize except that for the original query it will not use data. Thus if one wants to compare a model's theoretical value with the model's recorded value one should compare the values returned by forced_theorize and read_data.
10 Command-line operation

The Scientific Metareasoner has the commands for
1. querying the ontology,
2. querying the knowledge base more generally,
3. searching model space directly and through simulators,
4. changing the reasoner's behavior, and,
5. miscellaneous commands.

All commands should take parameters (if any) in parenthesized, comma-separated lists. All commands end with periods (".").

As the ontology is stored in metaphysics, the ontological commands are a special case of more general knowledge base querying commands.

**list(object).** This command lists the ontological instance object. This command is most useful when used with the function **member(class),** which returns the members of class **class.**

**Figure 3-1: Usage of list(class):**

```
Sci-reason> list(member(allele_class)).
allele_class
   allele_a1
   allele_a2
   allele_b1
   allele_b2
   leopard_frog_coloring_gene_allele_b
   leopard_frog_coloring_gene_allele_plus
Sci-reason>
```

**subclasses(class).** Lists the subclasses of class **class.**

**Figure 3-2: Usage of subclasses(class):**

```
Sci-reason> subclasses(chemical_fragment_class).
   chemical_fragment_class
   chemical_class
   allele_class
   allele_pair_class
Sci-reason>
```

**superclasses(class).** Lists the superclasses of class **class.**

**Figure 3-3: Usage of superclasses(class):**

```
Sci-reason> superclasses(chemical_fragment_class).
   chemical_fragment_class
   physically_manifested_patterns_class
   empirical_entity_class
```
everything
Sci-reason>

instances(class). Lists the instances of class class. (Like list(member(class)).)

Figure 3-4: Usage of instances(class):

Sci-reason> instances(allele_class).
  allele_a1
  allele_a2
  allele_b1
  allele_b2
  leopard_frog_coloring_gene_allele_b
  leopard_frog_coloring_gene_allele_plus
Sci-reason>

classes(instance). Lists the classes of which instance instance is a member.

Figure 3-5: Usage of classes(instance):

Sci-reason> classes(allele_a1).
  allele_class
  chemical_fragment_class
  physically_manifested_patterns_class
  empirical_entity_class
  everything
Sci-reason>

The more general knowledge-querying commands are used to ask the all components for an answer.

determine(object_list,attribute).

determine(object_list,attribute,comp_approach). Tries to determine the value(s) of attribute attribute for the objects of object_list. Object list may be just one object or may use the member(class) function. If the computational approach is not explicitly given as parameter comp_approach it defaults to theorize. If determination of the value is successful it returns both the value and the component that told it the final answer.

Figure 3-6: Usage of determine(object_list,attribute):

Sci-reason> instances(gene_class).
  gene_a
  gene_b
  leopard_frog_coloring_gene
Sci-reason> determine(member(gene_class),genes_allele_attr).
  gene_class
  gene_a at universal_state:
  genes_allele_attr: value(allele_a1,gene_a,genes_allele_attr) (data_reason_meth)
  gene_b at universal_state:
genes_allele_attr: value(allele_b1,gene_b,genes_allele_attr) (data_reason_meth)

leopard_frog_coloring_gene at universal_state:
genes_allele_attr: <no-prediction>

Sci-reason>

compare(object_list,attribute).
compare(object_list,attribute,comp_approach1).
compare(object_list,attribute,comp_approach1,comp_approach2).

resolve(sentence,component).

Model search and simulation commands.
justify(component).

process_instance_restruct(component,proc_inst).

compare_metaphysics(file).

The reasoning system has several commands that change its behavior.
context.
context(context). The context serves to update the knowledge base at query time. For example, to do run the same query two times under slightly different assumptions one would place the first assumption in the context, run the query, place the second assumption in the context, and then re-run the query.

The context command with no parameters tells the current context that all queries will use. The context command with parameter context specifies makes the system use context as the new querying context.

There is a special syntax for context. It is a square-bracketed, comma-separated Prolog list of variable-less predicates. The only useful predicates are property(object,attribute,value) which means that (for the purposes of this query) assume the object object has attribute attribute value equal to value. There is an abbreviated form for values of the context itself. The predicate prop(attribute,value) which means that (for the purposes of this query) assume the context has attribute attribute value equal to value.

The state is a special portion of the context. It may be altered with the context command or with the more specialized state command.

The default value for context is [prop(state_attr,universal_state)].

state.
state(state). The state is a special portion of the context that helps to specify queries. All values have a state that tells the time and duration that the value holds. Further,
all queries have an associated state so the system knows to look for matching values.

The state command with no parameters tells the system to return the current state to be used in queries. The state command with parameter state tells the system to accept the parameter as the new state to use in queries.

The default state is universal_state which encompasses all space and time. Most empirical values pertain to some proper subset of this, so state should be changed to the state of interest before empirical queries are submitted.

approach.
approach(approach).
approach(approach,defn). The command with no parameters prints out the currently-defined computational approaches (order in which components are checked to answer queries). The command with one parameter approach tells the order of components for approach approach. The command with two parameters redefines approach to check the components in the order specified by defn. Parameter approach must not be one of the fixed-definition approaches (given, ab_initio, read_data, theorize, empiricize or forced_theorize). These fixed-definition approaches cannot be overwritten.

set.
set(attribute).
set(attribute,value). The command with no parameters causes the system to print the names of all system attributes, their current values, and the possible values. The command with one parameter attribute tells the current value of the system attribute and its possible values. The command with two parameters sets the system attribute attribute to the have value value.

The system attributes are:
suggestion_mode (Values {on,off}. Defaults to on.) If set to on this attribute causes the system to give you suggestions as it sees fit. For example, if a determine command fails and the state is not set then it would tell you to check the state.
trace (Values {on,off}. Defaults to off.) If set to on then this attribute causes the system to have the user manually trace through the Scilog code during determine and compare commands. This is meant for debugging, and is tedious.
verbose (Values {on,off,empirical_only}. Defaults to off.) If set to on then this attribute causes the system to print out where it goes in the Scilog code without the user having to manually step through the code. If set to empirical_only then is only prints out its progress in the empirical components (theory, law and data).

By default this output goes to C's stdout file. It may, however, be redirected to a file by using the VERBOSE_FILE=<filename> option on the
command line when the program is first started. The output can be very long and tedious to search through.

**application_verbose** (Values {**on**, **off**}. Defaults to **off**.) If set to **on** then this attribute causes the system to print out its progress in areas other than Scilog querying. For example, it can be used to examine the reasoning done by the system when doing a **compare_metaphysics** command.

The command is like **verbose** in that by default its output goes to C’s **stdout** file. The output may, however, be re-directed to a file by using the **VERBOSE_FILE=<filename>** option on the command line when the program is first started.

Finally, the reasoner has miscellaneous commands to round out its functionality.

**help.**

**help(command).** The command with no parameters causes the system to briefly introduce all commands. The command with one parameter **command** prints a more detailed description of **command**.

**end.** Ends the reasoner session and returns control back to the operating system.