

# A Scientific Culture, Computer Language and Computing Environment for Science: Introducing `std_2009A_PhillipsJ`, `SciRep 7B`, and the `Scienceomatic 7B`

*Joseph Phillips*

*2009 January*

**Abstract.** This document serves as a users' guide for three entities: the scientific computing culture named “`std_2009A_PhillipsJ`”, the programming language for science named “`SciRep 7B`”, and the computing environment for science named “`Scienceomatic 7B`”. All three were designed for broad application to the sciences: from small scale high-energy physics to large scale relativity, and from the mechanisms of chemistry to the historical sweep of evolutionary biology.

## 1. Introduction

### ***1.1 About `std_2009A_PhillipsJ`, `SciRep 7B`, and the `Scienceomatic 7B`***

Many computer languages and computing environments exist, but none to my knowledge has been designed solely to capture the full breadth of scientific knowledge. The aim of this work is to change that.

The `Scienceomatic 7B` is an unrepentedly named computing environment that supports scientific representation, reasoning, discovery and visualization. To get most real work done in it one must tell it what to believe, and what to do.

The `Scienceomatic's` programming language is called `SciRep`, which is now also at version 7B. However, many scientists share a *large* body of common beliefs. Requiring each new user to re-enter these beliefs for him- or herself needlessly wastes effort.

The `Scienceomatic 7B` by default loads an initial program called `std_2009A_PhillipsJ` that hopefully takes away most of the drudgery of defining the common things. This program is called a “scientific culture” for reasons explained in the next paragraphs. And this “culture”, dear reader, allows you to jump write in and tailor your own knowledge base.

But first, a bit more on the distinctions among `Scienceomatic`, `SciRep`, `std_2009A_PhillipsJ`, and other concepts with which you may be familiar. The distinction between the *environment* `Scienceomatic` and the *language* `SciRep` is a subtle one, and one which it sometimes *might* be safe to ignore<sup>1</sup>. The `Scienceomatic` supports `SciRep`, and to the extent that the former is partially implemented in the latter the two are as inseparable as Unix (tm) is from C, or perhaps more so. Indeed the analogy of operating system/privileged programming language roughly describes the relationship between `Scienceomatic` and `SciRep`.

---

<sup>1</sup> Or it might not. Time will tell.

We call `std_2009A_PhillipsJ` a “scientific culture” because we believe the English word “culture” is the closest common word that captures its semantics. What `std_2009A_PhillipsJ` really is is the specification of a series of files to load into the `Scienceomatic`. These files tell the environment about the environment itself, about things that scientists worldwide of the early 21<sup>st</sup> century commonly discuss (*e.g.* the Earth, *H. sapiens*, carbon atoms, *etc.*), and about concepts that scientists worldwide of the early 21<sup>st</sup> century use to describe the things of the previous category (*e.g.* meters, time, *etc.*). The word “culture” often assumes a shared language or a shared historical experience. For us, `SciRep` written in early 21<sup>st</sup> century American<sup>2</sup> scientific<sup>3</sup> English is that language, and the historical experience is the development of the commonly-held beliefs of modern scientists until the early 21<sup>st</sup> century.

Computer science has the word “library” for a set of files generally provided by the authors of an environment, that are not part of the environment proper, and that may be tinkered with by expert users. By this definition `std_2009A_PhillipsJ` is a library, but we believe it to be something more. Just as judging one culture to be “objectively” inferior to another is sociologically suspect, so too we expect any pronouncement of an “optimal” scientific culture file scheme would be met with extreme skepticism. Contrast that to a traditional computer science library, where replacing a  $O(n)$  search algorithm with a  $O(\log_2 n)$  would be non-controversial. Indeed, `std_2009A_PhillipsJ` is provided solely as a convenience. It is *hoped* that you will extend it, *expected* that you will optimize it for your application, and *fervently desired* that scientists worldwide will participate in improving it and periodically updating it.

Over the years philosophers of science have developed terms that better approximate the fluid notion of “scientific culture” as described above. Thomas Kuhn defined the word “paradigm” to encompass the body of shared beliefs about scientific objects, how to interpret data, and what phenomena are worth explaining. Imre Lakatos defined the term “research programme” to mean the statements of Kuhn's paradigm separated into two bodies: a “hard core” of unshakable beliefs (“unshakable” in the sense that if/when they are refuted then the whole research programme has to be abandoned), and a set of “auxillary hypotheses” that can be more readily modified to fit new data. Larry Laudan introduced the term “research tradition” to cover the people (not necessarily the statements) that held similar beliefs and solved problems in similar fashions.

All three terms (“paradigm”, “research programme” and “research tradition”) cover the expected evolving nature of what we call a “scientific culture” better than the computer science term “library”. Still, we choose to use the term culture rather than reuse one of these other terms. In English the word “culture” can be hierarchical. Within a given culture there may exist several subcultures, and within a given subculture there may be several sub-subcultures. While one “paradigm” or “research programme” or “research tradition” may share some beliefs with one of its competitors, these terms are generally meant to contrast one set of scientists (and/or their beliefs) with another set. This differs from `std_2009A_PhillipsJ`, which is meant to be a shared starting point for most scientists of the early 21<sup>st</sup> century. It is expected that biologists will take it and extend it to form the subculture `biology_2012A_worldwide`, and that evolutionary biologists will take that subculture and extend it to form the sub-subculture `evobiology_2014B_worldwide`. Whether or not such sub-subcultures can be re-combined into one master `std_2020_worldwide` culture again is an open question.

By the way, artificial intelligence has the term “knowledge base” that captures the declarative nature of the representation and the potentially diverse nature of what is included. This term perhaps comes

---

2 American, so “characterize” instead of “characterise”.

3 Scientific, so “kilometers” instead of “miles”.

the closest to our “scientific culture”. However, what is missing from “knowledge base” is the expectation of dynamism, cumulative growth and community effort implied by the word “culture”. That said, any meaningfully-sized knowledge set that you load into the Scienceomatic is a knowledge base, and the Scienceomatic keeps one composite knowledge base in its memory. Thus, all scientific cultures written in SciRep 7B (including std\_2009A\_PhillipsJ) are knowledge bases; but not all knowledge bases are scientific cultures.

## 1.2 How the standard culture, language and environment were designed

Now that we have defined the environment, language and “scientific culture” we can discuss some of the design principles that went into all three:

1. **The ultimate purpose of the environment, language and scientific culture is to facilitate the search for extensions to knowledge bases and for explanations.** It is assumed that one wants to make their knowledge base more encompassing, more accurate, less self-contradictory, and more redundantly self-supportive. The computer language Prolog was used as the model for the behavior of a re-query-able, deductive search environment for parameterized solutions.
2. **The language natively supports descriptions of scientific things, especially processes.** One of the great achievements of the field of computational scientific discovery (“CSD”) is the realization that processes are a basic type of “scientific thing” besides the obvious Earth, *H. sapiens*, and carbon atoms. Processes (e.g. acceleration in a uniform gravitational field,  $S_N2$  reactions, and speciation) are a *basic* scientific thing, but not a *primitive* scientific thing. Languages for science must support ways of talking about the composition of processes from more primitive processes, into more complex processes, and their rates of change. Thus, languages must have a sufficiently rich and unified manner for speaking of process composition and differential equations.
3. **The language is not just a language of scientific things but also of scientific explanations.** Two levels of explanations are supported. An *explanatory trace* is a description of a path of the reasoning taken through the knowledge base to answer a query. It can be thought of as an answer to a “why” question. An *explanatory story* combines one or more explanatory traces and/or explanatory stories into a (hopefully) coherent explanation of a partial ordering of them in terms of believability. Explanatory stories are roughly analogous to academic conference papers: narratives telling what was tried, what succeeded, what failed, and why.
4. **The environment and language handles the messy data details.** The Scienceomatic and SciRep automatically handle dimension checking, unit conversion, sanity checking of values (e.g. “Is that mass value negative?”), and the proper use of values their given precision. This distinguishes SciRep from most other computer languages, which would subcontract such hassles to some “library” (in the aforementioned computer science definition of the word). SciRep has no “primitive” type like `int` found in C, C++ and Java. SciRep's most primitive datatype is `value`, which automatically carries not just an `int` or `double` primitive value but also precision, duration, subject, attribute, domain, dimension, units and axis information.
5. **The environment and language use a unified class hierarchy.** Everything the Scienceomatic knows of falls under one of four categories: (1) `empiricalEntitySet` for representing scientific things like the Earth and  $S_N2$  reactions; (2) `culturalConventionSet` for representing concepts used for discuss the things of `empiricalEntitySet`, like meters and time; (3) `programObjectSet` for

representing the state of the environment itself; and (4) `attributeSet` for talking about members of the other three. `SciRep` gets its idea for a single unified knowledge hierarchy from `Java`.

6. **The environment and language are designed for large and jointly-authored knowledge bases.** One example of this is the support for namespaces, which was taken from `C++` and `XML`. Another example is the support for *authorship*: the ability to attribute a set of people, a time, and a reason, to most of what is in the knowledge bases.
7. **Reuse of existing computation infrastructure in wide use.** The `Scienceomatic 7B` environment makes use of the Java Virtual Machine (JVM) for universalness. The environment also makes use of `Lisp` for its ability to dynamically define functions, and for its large base of artificial intelligence programs and programming talent.
8. **In general computation is done in an exhaustive, “brute-force”, breadth-first-search like manner.** We assume that the user turned to a computer in the first place for its speed and its ability to try all options. Heuristics can potentially speed computation, but they may or may not miss something. Breadth-first searches yield the shortest explanatory traces first, and we make the Occam's razor-like assumption that “shortest explanation”  $\approx$  “simplest explanation”. In practice, we expect the `Scienceomatic 7B` architecture to use more space efficient searches that are equivalent to breadth-first search, like iterative-deepening depth-first search.
9. **Symbolic computation is favored over numeric.** In general `Scienceomatic 7B` values distributions. Favoring symbolic computation over numeric potentially speeds computation by using integer and pointer operations instead of floating point ones. It also potentially fights the “distribution size bloat” that results when multiple binary operations are applied to values with distributions. Additionally, at the level of processes, relying on qualitative reasoning defines helps to define the search space that numeric operations have to consider.

In terms of its overall style `SciRep` takes after `Java`, but has some important differences. One superficial difference is that the operator `new` is not required to make a new object. Like `Java`, all objects are stored on the heap. Unlike `Java` one can just say the class' name in a special structure that encloses the parameters.

A larger difference is that it take object orientation to a more extreme level. `Java` has primitive values (`int`, `long`, `double`, `char`, *etc*). `SciRep` has just one: the `value` structure. This structure may hold data of a variety of primitive types including integer, rational, floating point and string.

A third difference is the importance of assertions over member variables. `SciRep` objects do not have members variables, but they do have assertions that are intended to help compute one or more . . .

A fourth difference is the importance of the built-in computation mechanism over methods.

(How to build user-defined types? Cons? Struct? Methods including constructors?)

(For standard library reuse object or source code? I think object)

## 2. Basic Representation and Calculation with the value structure

### 2.1 Getting Started

So let's jump in! The syntax for starting the `Scienceomatic` environment is:

```
YourOSPrompt> som [-kb <kbFile>] [-web <portNumber>]
```

or perhaps:

```
YourOSPrompt> scienceomatic [-kb <kbFile>] [-web <portNumber>]
```

The parameter specifier for knowledge base files is `-kb` which precedes the name of a knowledge base file. The `-kb` must precede each knowledge base file to read, and they are all read in their listed order. The parameter specifier for the web interface is `-web` which precedes the name of the web server. The text of `<portNumber>` must be a port to connect to the local computer like 8080. Both the `-kb` and `-web` specifiers are optional; if `-kb` is not given then it will try to load `std_2009A_PhillipsJ` and if `-web` is not given then it will just bring up the command line interface.

It's that easy! Now you are running the environment. If you brought up the web interface then please open your web browser to connect to it.

Please take a moment to look around. You may see just the command prompt (perhaps in its own window). You may see the web interface with the expected menus `File`, `Edit`, `View`, ... `Help`. You may see both. Fundamentally all `Scienceomatic` commands are text-based commands: the web interface provides an easier interface to the underlying text-based system. Anything that you can do with the web you may do from the command line, and most of what you can do from the command line may also be done from the web interface.

We can use the environment as a basic calculator, as many other interpreted environments may be used. Using the web interface please go to `Calculate -> Expression`, or from the command line interface please just type:

```
SoM7B :) 4 * 8
```

The environment returns:

```
32
```

If you type:

```
SoM7B :) 4 + 2 * 8
```

you will get:

which is  $4 + (2 * 8)$ , instead of  $(4 + 2) * 8$  which would yield 42. The `Scienceomatic` has the typical precedence rules for arithmetic that other languages have. Of course, typing:

```
SoM7B :) (4 + 2) * 8
```

does yield 42.

Yeah, yeah . . . your accessory's calculator can do that. But can it do this?

```
SoM7B :) {value 200 kmDom} / {value 2 hoursDom}
{value 100 kmPerHoursDom}
```

The `Scienceomatic` “knows” what kilometers are, “knows” what hours are, and “knows” that dividing one by the other gives a velocity. But wait, there's more! Please try:

```
SoM7B :) cos({value 60 degreesDom})
{value 0.707106781187 negOneToOneDom}
```

Check it out: we specified the domain (and implicitly the units) of the value to give to the `cos()` function. In return we get the value 0.707106781187 *and* its corresponding domain, `negOneToOneDom`. Thus, the `Scienceomatic` is smart enough to know that any value returned by the `cos()` function must be between -1 and 1.

If you prefer radians over degrees the `Scienceomatic` can handle that too:

```
SoM7B :) cos({value (pi*0.25) radiansDom})
{value 0.707106781187 negOneToOneDom}
```

Well, what happens if you do *not* specify the domain?

```
SoM7B :) cos({value (pi*0.25)})
ERROR:
Function cos() requires a value with dimensionality angle. :(
```

Hmm, what happens if you mismatch the domains?

```
SoM7B :) 4 + {value 4 kmDom}
ERROR:
Function + requires arguments with the same dimensionality. :(
```

You *must* specify the *proper* domain. A hassle? Yes, but it keeps you honest. No more forgetting where `cos()` takes radians or degrees because it takes *both*!

***Let us review:***

1. The `Scienceomatic` can do common arithmetic.
2. The `Scienceomatic` keeps track of the domain of values. Domains keep track of a number of things including the dimensions, units, and legal ranges of the values that they modify.
3. The `Scienceomatic` uses these domains and complains if they do not match.
4. There are built-in constants, like `pi`.

## 2.2 The `value` structure's basics

You may create numeric constants by just typing an integer value like 4, or a floating point number like 0.25. Doing so will create a number with dimension `dimensionless` and units `unitless`. To create a number with dimensions and units other than that use the `value` structure. A simplified syntax is:

```
{ value <constant> [<domain>] }
```

The `value` structure is more powerful than this, and can also specify precision information, a state, a described subject, and a described attribute. However, for now, let us understand the domain.

For the `value` structure the keyword `value` must come first. It must<sup>4</sup> be followed by a constant of some kind. The domain is optional. If it is not given then the value becomes dimensionless and unitless.

By the way, `SciRep` creates instance of objects with structures. All structures have the syntax:

```
{ <structureName> [param1] [param2] . . . [paramn] }
```

Please note that the name of the structure to create must come first, and that the parameters (if there are any) use no commas or any other symbol for separation.

You may create your own domains, and domains are automatically created for you as you work, but the culture `std_2009A_PhillipsJ` has several built in domains for you already. Here are some of the more useful ones listed by their dimension:

| <b>Dimension:</b>           | <b>Domains with that dimension:</b>   |
|-----------------------------|---|
| <code>dimensionless</code>  | <code>dimensionlessDom, negOneToOneDom</code>                                   |
| <code>count</code>          | <code>widgetCountDom</code>   |
| <code>linearPosition</code> | <code>metersDom, kmDom, cmDom, mmDom, micronDom, nmDom, angstromDom</code>      |
| <code>angle</code>          | <code>degreesDom, neg180To180DegreesDom, radiansDom, negPiToPiRadiansDom</code> |
| <code>time</code>           | <code>secondsDom, hoursDom, gregorianDom</code>                                 |
| <code>mass</code>           | <code>gramsDom, kgDom</code>  |
| <code>charge</code>         | <code>coulombsDom</code>  |
| <code>area</code>           | <code>sqrMetersDom</code>   |

<sup>4</sup> Or by a list of constants, but we get ahead of ourselves.

|              |                                 |
|--------------|---------------------------------|
| volume       | cubicMetersDom                  |
| velocity     | kmPerHourDom, kmPerSecDom       |
| acceleration | kmPerHourSqrDom, kmPerSecSqrDom |
| force        | newtonsDom                      |
| energy       | joulesDom, caloriesDom          |
| pressure     | pascalsDom, megaPascalsDom      |

For a full list of the domains and their properties that `std_2009A_PhillipsJ` defines, please see **Appendix I**.

Domains have several properties, including dimension and units. They are accessible with the attributes `domainsDimensionAttr` and `domainsUnitsAttr` respectively:

```
SoM7B :) metersDom->domainsDimensionAttr
linearPosition
```

```
SoM7B :) metersDom->domainsUnitsAttr
meters
```

As you see, the operator `->` returns the attribute specified by the right-hand-side of the object given on the left-hand-side. An attribute that describes aspect `<aspect>` of objects `<object>` generally have morphology `<object>s<aspect>Attr` or `<object>es<aspect>Attr`, where the “s” or “es” approximates the English possessive “'s” or “s'”.

One may use the attribute `valuesDomainAttr` to retrieve information on values:

```
SoM7B :) {value 100 kmPerHoursDom}->valuesDomainAttr
kmPerHoursDom
```

```
SoM7B :) {value 100 kmPerHoursDom}->valuesDomainAttr->
domainsDimensionAttr
velocity
```

```
SoM7B :) {value 100 kmPerHoursDom}->valuesDomainAttr->
domainsUnitsAttr
kmPerHours
```

```
SoM7B :) 90->valuesDomainAttr->domainsUnitsAttr
unitless
```

The last one shows that even dimensionless constants really are value structures too.

**Let us review:**

1. The value structure creates value objects. Its most basic syntax is `{value <constant> [<domain>]}`.
2. The operator `->` gets the value of the specified object's specified attribute.

3. The attribute `valuesDomainAttr` maps from a value to its domain.
4. The attribute `domainsDimensionAttr` maps from a domain to its dimension.
5. The attribute `domainsUnitsAttr` maps from a domain to its units.

## 2.3 Numeric operators of value structures

Now we can handle the basics of the value structure. Let us see what operations we can do. We know that this is illegal.

```
SoM7B :) 4 + {value 4 kmDom}
ERROR:
Function + requires arguments with the same dimensionality. :(
```

But let us try the following two:

```
SoM7B :) {value 4 metersDom} + {value 4 kmDom}
{value 4004 metersDom}

SoM7B :) {value 4 kmDom} + {value 4 metersDom}
{value 4.004 kmDom}
```

On one hand that is kind of cool: in both cases it was smart enough to convert the units (kilometers to meters in the first case, meters to kilometers in the second). However, there is perhaps something disconcerting about this. *Addition is no longer commutative!* The domain used by the sum comes from the first operand, not the second. And, as  $a+b+c+d$  is computed in a left associative manner as  $((a+b)+c)+d$  the domain used by the resulting whole sum always comes from  $a$ . Thus we have:

```
SoM7B :) {value 4 kmDom} + {value 4 metersDom}
{value 4.004 kmDom}

SoM7B :) {value 4 kmDom} + {value 4 metersDom} + {value 4 mmDom}
{value 4.004004 kmDom}

SoM7B :) {value 4 kmDom} + {value 4 metersDom} + {value 4 mmDom} +
         {value 4 micronsDom}
{value 4.004004004 kmDom}

SoM7B :) {value 180 degreesDom} + {value (pi) radiansDom}
{value 360 degreesDom}
```

By the way, in the last example the constant `pi` must be enclosed in parentheses. This forces it to be computed into 3.14159265359..., and for that value to be sent to the value structure. Otherwise, the value structure would not know what to do with just `pi`.

Is this so bad? No, and yes. “No” in the sense that the Scienceomatic knows that the *fundamental* values of `{value 4004 metersDom}` and `{value 4.004 kmDom}` are “equal” by doing unit conversion:

```
SoM7B :) {value 4.004 kmDom} == {value 4004 metersDom}
true
```

“Yes” in the sense that while the fundamental values may be equal, the *whole* values are not:

```
SoM7B :) {value 4.004 kmDom}->valuesDomainAttr == {value 4004
metersDom}->valuesDomainAttr
false
```

The subtraction operator (-) also use the domain of the left most argument in the overall result:

```
SoM7B :) {value 4 kmDom} - {value 4 metersDom}
{value 0.0 kmDom}
```

This is because for both addition and subtraction the left operand is the *defining quantity*. Besides the domain, the defining quantity also gives the state, subject and attribute to the result, as we will see later.

The Scienceomatic also has multiplication (\*) and division (/). In general both of them yield a completely different domain than either of their arguments.

```
SoM7B :) {value 0.004 kmDom} * {value 4 metersDom}
{value 16 metersSqrDom}
```

```
SoM7B :) {value 4 metersDom} / {value 0.004 kmDom}
1
```

In the last example the resulting 1 is an abbreviation for {value 1 dimensionlessDom} obtained by the canceling of the length dimension in both the numerator and denominator.

There is also raising-to-the-power-of (^), whose second parameter (the exponent) must be dimensionless. If the second parameter (the exponent) is anything other than an integer or rational number than the first number (the base) must also be dimensionless.

```
SoM7B :) 6^2
36
```

```
SoM7B :) 6^{value 2 metersDom}
ERROR:
Operator ^ requires a dimensionless exponent 2nd parameter :(
```

```
SoM7B :) {value 6 metersDom}^2
{value 36 metersSqrDom}
```

```
SoM7B :) 6^pi
278.37757775
```

```
SoM7B :) {value 6 metersDom}^pi
ERROR:
Operator ^ requires an integer or rational exponent 2nd parameter
```

if the 1st parameter is not dimensionless :(

There are also the typical trigonometric unary functions: `sin()`, `cos()`, `tan()`, `asin()`, `acos()` and `atan()`. The first three must have arguments with domains with dimension angle. They return dimensionless values.

```
SoM7B :) sin({value (pi/2) radiansDom})  
1
```

```
SoM7B :) sin({value (pi/2)})  
ERROR:  
Function sin() requires a value with dimensionality angle. :(
```

The functions `asin()`, `acos()` and `atan()` must have arguments with domains with dimension dimensionless. They return a value with domain `radiansDom`.

```
SoM7B :) asin(1)  
{value 1.5707963268 radiansDom}
```

```
SoM7B :) asin({value 180 degreesDom})  
ERROR:  
Function asin() requires a value with dimensionality dimensionless.  
:(
```

If you want `angleDom` instead of `radiansDom` use the `rescale()` function whose syntax is `rescale(<value>, <desiredDomain>):`

```
SoM7B :) rescale(asin(1), degreesDom)  
{value 180 degreesDom}
```

```
SoM7B :) rescale({value 400 metersDom}, kmDom)  
{value 0.4 kmDom}
```

```
SoM7B :) rescale({value 400 metersDom}, degreesDom)  
ERROR:  
Domains metersDom and degreesDom have different dimensions. :(
```

The two last examples shows that `rescale()` works with other dimensions, within the constraints of legality.

SciRep also has a rich set of unary exponential and logarithmic functions: `log()` and `exp()` for base e, `log2()` and `exp2()` for base 2, and `log10()` and `exp10()` for base 10. All six require dimensionless arguments and return dimensionless results.

```
SoM7B :) log(exp(5))  
5
```

```
SoM7B :) log2(256)  
8
```

```
SoM7B :) exp10(4)  
1000
```

The last of the mathematical operators are unary negation (-), which negates numbers; unary positive (+), which return the same number that they were given; `floor()`, which returns the largest integer not greater than the argument; `ceil()`, which returns the smallest integer not less than the argument; and `abs()`, which returns the absolute value of its argument. All five return values with domains with the same dimensions as their arguments.

## 2.4 Variables and assignments

SciRep naturally has variables and assignment. Assignment is done with the assignment operator `:=`. Like in the programming language Prolog, variables do not have to be declared but they do have start with an uppercase letter. They are case sensitive.

```
SoM7B :) Result := 1000  
1000
```

```
SoM7B :) Result  
1000
```

```
SoM7B :) RESULT  
ERROR  
Unknown variable RESULT :(
```

## 2.5 A chance for the impatient to skip ahead

You now know enough about domains to move on to the third chapter on objects, if you are truly impatient. There is, however, a richness to `value` structures that we have yet to cover. So you may skip ahead to the next chapter, or continue reading in this one, or skip ahead but come back as needed.

The choice is yours.

## 2.6 Precision in `value` structures

So far we have seen `value` structures with single primitive values. However, `value` structures can store multiple sample values in explicit lists. Lists have square bracket delimited, comma-separated items like:

```
SoM7B :) {value [0.9, 1.0, 0.9, 0.85]}  
{value [0.85, 0.9, 0.9, 1.0]}
```

The same item may be present in the list multiple times. The items in the list may be given any order. When displayed, however, they are printed in canonical order by ascending value.

Conceptually the collection of values are stored in a data structures known as a *bag*. Bags are like lists in that they may contain the same item multiple times. Bags are like sets in that they are unordered. (They are *displayed* in ascending form, but not necessarily stored that way.)

SciRep has a number of attributes for accessing the sample values of a value structure. Attribute `valuesNumSamplesAttr` maps to an integer telling how many sampled values there are:

```
SoM7B :) {value [0.85, 0.9, 0.9, 1.0]}->valuesNumSamplesAttr
4
```

```
SoM7B :) 4->valuesNumSamplesAttr
1
```

Individual values are accessible with an array-style of access with `valuesSampledAt`:

```
SoM7B :) {value [0.85, 0.9, 0.9, 1.0]}->valuesSampledAt(0)
0.85
```

```
SoM7B :) {value [0.85, 0.9, 0.9, 1.0]}->valuesSampledAt(1)
0.9
```

```
SoM7B :) {value [0.85, 0.9, 0.9, 1.0]}->valuesSampledAt(2)
0.9
```

```
SoM7B :) {value [0.85, 0.9, 0.9, 1.0]}->valuesSampledAt(3)
1.0
```

```
SoM7B :) {value [170, 180, 190] degreesDom}>valuesSampledAt(1)
{value 180 degreesDom}
```

The argument to `valuesSampledAt()` must be a dimensionless non-negative integer less than the value returned by `valuesNumSamplesAttr`.

The array-style access that uses `valuesNumSamplesAttr` *cannot* be used to change sample values: once a value structure is defined it is immutable.

```
SoM7B :) {value [0.85, 0.9, 0.9, 1.0]}->valuesSampledAt(3) := 4.0
ERROR:
valuesNumSamplesAttr() cannot be used as an L-value :(
```

We will cover assignments in the next subchapter.

A single “representative” value of the sample value list can be returned with the attributes `valuesMeanValueAttr` (for the average value), `valuesMedianValueAttr` (for the middle value) and `valuesModeValueAttr` (for the most numerous value):

```
SoM7B :) {value [1, 1, 5, 6, 7]}->valuesMeanValueAttr
4
```

```
SoM7B :) {value [1, 1, 5, 6, 7]}->valuesMedianValueAttr
5
```

```
SoM7B :) {value [1, 1, 5, 6, 7]}->valuesModeValueAttr
1
```

Like all other value structures, multi-sample value structures can be operated on by either unary or binary operators. Unary operators just create a new value structure in which the operator was applied to each of the argument's sample values.

```
SoM7B :) sin({value [0, 5, 10] degreesDom})
{value [0, 0.0871557427477, 0.173648177667]}
```

```
SoM7B :) -{value [0, 5, 10]}
{value [-10, -5, 0]}
```

The behavior of multi-sample value structures operated on by binary operators is a little more complex. There are two case: when both operands ultimately stem from the same parent value, and when they do not.

When they do not, that is, when the binary operator operates on two different values from different sources, the following is done. First, let us assume the operator is addition, assume that the left hand side has m sample values, and assume that the right hand side has n sample values. If (m\*n) is less than or equal to the value of `system->systemsBehaviorObjAttr->maxSamplePopulationSize` then the operation will be done on each possible pairing of m left hand side sample values combined with n right hand side sample values. This will create a value structure with (m\*n) sample values (which may or may not be unique).

```
SoM7B :) {value [1, 2]} + {value [4, 5, 6]}
{value [5, 6, 6, 7, 7, 8]}
```

```
SoM7B :) {value [0.9, 1.0, 0.9, 0.85]} + {value [0.9, 1.0, 0.9,
0.85]}
{value [1.7, 1.75, 1.75, 1.75, 1.75, 1.8, 1.8, 1.8, 1.8, 1.85,
1.85, 1.9, 1.9, 1.9, 1.9, 2.0]}
```

The second example above shows that even if the value structures operated on are identical (in the sense of having the same numbers) one still gets this behavior.

If they come from different sources but (m\*n) would exceed `maxSamplePopulationSize` then (for addition) the resulting value structure would have m values generated by operating on each of the m left hand side sample values and the single “representative” (mean, median or mode) right hand side value. In the example below assume that `maxNumValuesSamplePts` is 8.

```
SoM7B :) system->systemsBehaviorObjAttr->maxNumValuesSamplePts :=8
8
```

```
SoM7B :) {value [1, 2]} + {value [4, 5, 6]}
{value [5, 6, 6, 7, 7, 8]}
```

```
SoM7B :) {value [0.9, 1.0, 0.9, 0.85]} + {value [0.9, 1.0, 0.9,
0.85]}
{value [1.7625, 1.8125, 1.8125, 1.9125]}
```

```
SoM7B :) system->systemsBehaviorObjAttr->maxNumValuesSamplePts :=
system->systemsBehaviorObjAttr->
defaultMaxNumValuesSamplePts
```

65

The first sum above resulted in only 6 sample values, less than or equal to 8, so all were generated. The second sum would have resulted in 16 sample values; clearly greater than 8. Thus the mean of the second value's sample values was obtained (0.9125) and it was added to each of the first value's sample values in turn.

Two natural questions are “*Why did you just tell us what happens for addition? And, why did addition take its values from its left hand argument?*” To address the first point, all binary operators (+, -, / and ^) preferentially use the left hand argument except for multiplication (\*), which uses the right. For all binary operators (except multiplication) the left hand side argument is called the *defining quantity*, the value which imparts most of the semantic content to the result. For multiplication, the right hand side quantity is the defining quantity. Defining quantities are discussed in more detail later in this chapter at subchapter \_\_.

All of the cases above dealt with two arguments that were completely unrelated. However, if both arguments derive from some common value then the Scienceomatic will try to recognize that commonality. It does so by keeping track of the derivation of value structures. Sample values do not multiply to the extent that they come from the same source value structure.

```
SoM7B :) X := {value [0.9, 1.0, 0.9, 0.85]}
{value [0.85, 0.9, 0.9, 1.0]}
```

```
SoM7B :) X + X
{value [1.7, 1.8, 1.8, 2.0]}
```

```
SoM7B :) X * 2
{value [1.7, 1.8, 1.8, 2.0]}
```

```
SoM7B :) Y := X
{value [0.85, 0.9, 0.9, 1.0]}
```

```
SoM7B :) X + Y
{value [1.7, 1.8, 1.8, 2.0]}
```

```
SoM7B :) Y := X * 2
{value [1.7, 1.8, 1.8, 2.0]}
```

```
SoM7B :) X + Y + Y
```

```
{value [4.25, 4.5, 4.5, 5.0]}
```

However, if variables X and Y are assigned differently then we will get different, more numerous results.

```
SoM7B :) X := {value [0.85, 0.9, 0.9, 1.0]}  
{value [0.85, 0.9, 0.9, 1.0]}
```

```
SoM7B :) Y := 2 * {value [0.85, 0.9, 0.9, 1.0]}  
{value [1.7, 1.8, 1.8, 2.0]}
```

```
SoM7B :) X + Y + Y  
{value [4.25, 4.3, 4.3, 4.4, 4.45, 4.45, 4.5, 4.5, 4.5, 4.5, 4.6,  
4.6, 4.85, 4.9, 4.9, 5]}
```

Once created value structures are immutable. However, if you would like to create one with few sample values use the `subsample(<value>, <newNumSamples>)` function. It returns a new value structure that has only `<newNumSamples>` of the sample values of `<value>`. The value given by `<newNumSamples>` must be a dimensionless integer between 1 and `<value>->valuesNumSamplesAttr` inclusive. It does the best job it can picking values from `<value>`'s explicit sample distribution evenly.

```
SoM7B :) subsample({value [0, 2, 4, 6, 8]}, 3)  
{value [0, 4, 8]}
```

Sometimes, rather than explicitly listing all values, it is more convenient to assume a canonical distribution. SciRep can represent both normal (“Gaussian” or “bell-curve”) distributions and uniform ones. The syntax for value structures with normal distributions is:

```
{value normal(<mean>, <stdDev>) [<param1>] . . . [<paramn>]}
```

where `<mean>` tells the mean value and `<stdDev>` tells the standard deviation. The value given for `<stdDev>` must be non-negative. There are two syntaxes for value structures with uniform distribution:

```
{value uniformEnds(<low>, <high>) [<param1>] . . . [<paramn>]}
```

```
{value uniformMid(<mean>, <halfWidth>) [<param1>] . . . [<paramn>]}
```

The first form (`uniformEnds`) takes the end points of the range, `<low>` being the lowest and `<high>` being the highest. The value given for `<low>` must not exceed that given for `<high>`. The second form (`uniformMid`) takes range's mean `<mean>` and half of the range's width `<halfWidth>` (which must be a non-negative number).

```
SoM7B :) {value normal(0,1)}  
{value normal(0,1)}
```

```
SoM7B :) {value normal(0,-1)}
ERROR:
Attempt to define a normally-distributed value with a negative
standard deviation :(
```

```
SoM7B :) {value uniformEnds(-1,1)}
{value uniformEnds(-1,1)}
```

```
SoM7B :) {value uniformMid(0,1)}
{value uniformEnds(-1,1)}
```

```
SoM7B :) {value uniformEnds(1,-1)}
ERROR:
Attempt to define a value with an illegal uniform range :(
```

The second from the last example shows that the default way to display a uniform range is with `uniformEnds`.

The Scienceomatic preserves these implicit ranges across operations as much as possible:

```
SoM7B :) {value normal(0,1)} + 4
{value normal(4,1)}
```

```
SoM7B :) {value normal(0,1)} / 2
{value normal(0,0.5)}
```

```
SoM7B :) -{value uniformEnds(-1,1)}
{value uniformEnds(-1,1)}
```

```
SoM7B :) 6 - {value uniformEnds(0,1)}
{value uniformEnds(5,6)}
```

```
SoM7B :) {value uniformEnds(0,1)} * 10
{value uniformEnds(0,10)}
```

However, it encounters an operation that does not result in either a normal or uniform distribution then it creates a derivative explicitly-sampled value structure with `system->systemsBehaviorObjAttr->numImplicitToExplicitConvertSamplePts` sample points.

## 2.7 Limits of domains

We have already seen cases where the domain seems to keep track of the limits of legal values. For example, the trigonometric functions `sin()` and `cos()` can only generate values between -1 and 1 inclusive, and this is reflected in domains of the values they return.

```
SoM7B :) cos({value 60 degreesDom})
```

```
{value 0.707106781187 negOneToOneDom}
```

Of course, neither `sin()` nor `cos()` can exceed these bounds. But also neither can you:

```
SoM7B :) {value 1.1 negOneToOneDom}
```

ERROR:

```
Attempt to create a value with a value outside of legal range
definitions for negOneToOneDom :(
```

Scienceomatic numeric domains have 6 types of bounds described by the 12 attributes of domains listed below:

1. Range definition limits. (Low: `domainsLoRangeDefineLimitAttr`. High: `domainsHiRangeDefineLimitAttr`.) The domain is logically defined to have these hard endpoints. *Example*: masses cannot be negative. The limits default to -infinity and +infinity respectively if not specified.
2. System limits. (Low: `domainsLoSystemLimitAttr`. High: `domainsHiSystemLimitAttr`.) The system being measured has these physical endpoints. *Example*: if we were interested in mass of some portion of the Earth-Moon system then then this mass cannot exceed  $mass_{earth} + mass_{moon}$ . Both of these limits default to their corresponding range-define limits if not specified.
3. Detection limits. (Low: `domainsLoDetectLimitAttr`. High: `domainsHiDetectLimitAttr`.) Recording instruments cannot detect values outside this range. *Example*: the magnitude of earthquakes too small to detect. Both of these limits default to their corresponding range-define limits if not specified.
4. Saturation limits. (Low: `domainsLoSaturateLimitAttr`. High: `domainsHiSaturateLimitAttr`.) The system empirically does not exhibit many values outside this range, even though in principle it could. *Example*: in a click-the-mouse-after-seeing-a-light experiment the fastest humans tend to be around 130 milliseconds [Human Benchmarks, 2006-2008]. Both of these limits default to their corresponding system-limits if not specified. The saturate-limits are *soft limits* and are not meant to be a definitive threshold.
5. Reliability limits. (Low: `domainsLoReliableLimitAttr`. High: `domainsHiReliableLimitAttr`.) Instruments cannot be relied upon reproducibly to detect values outside this range. *Example*: the mass of small near-Earth asteroids that are detectable because they are relatively close. Similarly sized objects in the Van Oort belt (beyond the orbit of Pluto) are not detectable. Both of these limits default to detect-limits if not specified. The reliable-limits are *soft limits* and are not meant to be a definitive threshold.
6. Observed limits. (Low: `domainsLoObservedLimitAttr`. High: `domainsHiObservedLimitAttr`.) No values were observed beyond these values. Both of these limits default to the more restrictive of their corresponding detect-limits and system-limits if they are not specified or computed.

Three classes of consistency constraints exist: **system constraints**, **instrument constraints** and **data constraints**. Their definitions are given in table below.

### Definitions of constraints on values in domains

| Constraint name | Constraints   |
|-----------------|---|
| system          | $loSaturateLimit \geq loSystemLimit \geq loRangeDefineLimit;$<br>$hiSaturateLimit \leq hiSystemLimit \leq hiRangeDefineLimit$                                   |
| instrument      | $loReliableLimit \geq loDetectLimit \geq loRangeDefineLimit;$<br>$hiReliableLimit \leq hiDetectLimit \leq hiRangeDefineLimit$                                   |
| data            | $loObservedLimit \geq loSystemLimit;$<br>$loObservedLimit \geq loDetectLimit;$<br>$hiObservedLimit \leq hiSystemLimit;$<br>$hiObservedLimit \leq hiDetectLimit$ |

The **reliable** and **saturate** are “soft” bounds and their semantics is open to domain-specific interpretation. **Detectable**, **system** and **logical** are “hard” bounds fixed by the recording device, domain knowledge and the definitions of attributes.

The `Scienceomatic` environment automatically generates an error and stops when:

1. A computed value exceeds range definition limits.

The `Scienceomatic` environment automatically generates a warning (but keeps going) when:

1. A computed value exceeds system limits, or
2. A computed value exceeds detection limits.

The generation of range definition error is automatic and cannot be overridden. However, the `Scienceomatic` may be told to generate errors and stop when other domain bounds are exceeded. The generation of the system and detection warnings is the default behavior. The environment may be told to generate (or not generate) warnings for any of the six limits *except* range definition (for which generating an error is the inalterable behavior).

```
SoM7B :) EarthMoonMassDomain := copy(kgDom) // Make new domain obj
domain198
```

```
SoM7B :) EarthMoonMassDomain->domainsUnitsAttr
kg
```

```
SoM7B :) EarthMoonMassDomain->domainsLoRangeDefineLimitAttr
{value 0 domain198}
```

```
SoM7B :) EarthMoonMassDomain->domainsHiSaturateLimitAttr
{value +inf domain198}
```

```
SoM7B :) EarthMoonMassDomain->domainsHiSystemLimitAttr :=
earth.physicalObjectsMassAttr +
moon.physicalObjectsMassAttr
{value 6.05e+24 domain198}
```

```
SoM7B :) EarthMoonMassDomain->domainsHiSaturateLimitAttr
{value 6.05e+24 domain198}
```

```
SoM7B :) SunsMass := sun->physicalObjectsMassAttr
```

```
{value 1.99e+30 kgDom}
```

```
SoM7B :) SunsMass := rescale(SunsMass,EarthMoonMassDomain)
```

```
WARNING:
```

```
Generated value outside of systems limit. :0
```

```
{value 1.99e+30 domain198}
```

Typing **EarthMoonMassDomain := copy(kgDom)** creates a new domain that is a copy of kgDom. This domain is internally named domain198, and we can either access it under that name or by our variable EarthMoonMassDomain. We can see that it copied units of kilograms, the lower range definition limit of 0 kgs from kgDom, and the higher saturation limit of +inf kgs.

Assigning **EarthMoonMassDomain->domainsHiSystemLimitAttr** to the sum of the Earth's and Moon's mass also sets **domainsHiSaturateLimitAttr** too. This was demonstrated by the saturation limit decreasing from +inf kgs to 6.05e+24 kgs.

We can use this new domain, too. Setting **SunsMass** to the mass of the Sun is no problem. However, attempting to use our new domain domain198 that corresponds to some fraction of the combined mass of the Earth and Moon to hold **SunsMass** generates a systems limits warning.

## 2.8 Subjects, attributes and defining operands

Scienceomatic value structures keep track of both the subject and attribute of the thing being described. The attribute valuesSubjectAttr maps from value structures to the thing they describe, and the attribute valuesAttrAttr maps from value structures to the aspect of the described thing.

```
SoM7B :) SunsMass := sun->physicalObjectsMassAttr
```

```
{value 1.99e+30 kgDom}
```

```
SoM7B :) SunsMass->valuesSubjectAttr
```

```
sun
```

```
SoM7B :) SunsMass->valuesAttrAttr
```

```
physicalObjectsMassAttr
```

When you create a value structure you may specify both the subject and the attribute by preceding either with the labels subject: and attr: respectively.

```
SoM7B :) Radius := {value 6.37e+3 kmLengthDom
```

```
subject: earthsEquivalentSphere
```

```
attr: physicalObjectsRadiusAttr }
```

```
{value 6.37e+3 kmLengthDom}
```

```
SoM7B :) Radius->valuesSubjectAttr
```

```
earthsEquivalentSphere
```

```
SoM7B :) Radius->valuesAttrAttr
```

```
physicalObjectsRadiusAttr
```

Both a subject and an attribute are computed during operations. The Scienceomatic keeps track of the derivation of attributes, so it knows, for example that squaring a radius of an object gives you an “area” of that object, and multiplying that by another radius yields a “volume”:

```
SoM7B : ) Area := 4/3*pi*(Radius*Radius*Radius)
{value 1.08e+12 kmAreaDom}
```

```
SoM7B : ) Area->valuesSubjectAttr
earthEquivalentSphere
```

```
SoM7B : ) Area->valuesAttrAttr
physicalObjectsAreaAttr
```

Of course, the Scienceomatic is looking at the dimensionality and the attributes, and not at the form of the formula. If the formula is wrong the attribute might be right but the numeric value could be wrong.

```
SoM7B : ) Area := 4/3*pi*(Radius*Radius*Radius)*1000
{value 1.08e+15 kmAreaDom}
```

```
SoM7B : ) Area->valuesSubjectAttr
earthEquivalentSphere
```

```
SoM7B : ) Area->valuesAttrAttr
physicalObjectsAreaAttr
```

In binary expressions the operand that give the subject and state of the resulting value is called the *defining operand*. For additions (and subtractions, divisions, modulus, and raising-to-the-power-of) the defining quantity is the left-hand-side operand while for multiplication the defining quantity is the right-hand-side operand. This means that neither addition nor multiplication are commutative. Operations on values are *defined* liked this. Users of SciRep must always bear this in mind.

The proper way to right a SciRep expression is for (for addition, subtraction, division, modulus and raising-to-the-power-of) the defining operand to be on the left-hand-side while for multiplication it must be on the right-hand-side.

Some familiar equations will have slightly different forms:

#### The Ideal Gas Law:

Physics notation:  $PV = nRT$

SciRep notation:

```
Gas->gasesPressureAttr
* Container->objectsInternalVolumeAttr
= {value normal(8.3145,0.00005) joulesPerMoleKelvinDom}
* Gas->materialsMoleNumAttr
* Gas->objectsTemperatureAttr
```

In SciRep prose we write “ $PV=RnT$ ” instead of the more familiar “ $PV=nRT$ ” because the

constant  $R$  should go first: being constant it gives us the least amount of information about any particular sample of gas at any particular time.

Newton's Second Law (special case when  $\Delta m = 0$ ):

Physics notation:  $F = ma$

SciRep notation:

```
Object->forceAttr =
Field->AccelerationAttr * Object->physicalObjectsMassAttr
```

In SciRep prose we write “F=am” instead of the more familiar “F=ma” because the acceleration term should go first because it might depend on the gravitational/electrical/magnetic/etc. field instead of the object itself. It gives us the least amount of information about any particular object at any particular time.

When doing algebra internally SciRep manipulates expressions so that the defining quantity holds or is appropriately transferred. In the transformations below terms/factors  $d_1$  and  $d_2$  denote the defining quantities:

|                     |                     |                      |                               |
|---------------------|---------------------|----------------------|-------------------------------|
| $d_1 +/- f() = d_2$ | subtract +/-f() ==> | $d_1 = d_2 -/+ f()$  | (NOT $d_1 = -/+f() + d_2$ )   |
| $d_1 +/- f() = d_2$ | subtract $d_1$ ==>  | $+/-f() = d_2 - d_1$ | ( $d_2$ dominates)            |
| $f() * d_1 = d_2$   | divide by f() ==>   | $d_1 = d_2 / f()$    | (NOT $d_1 = d_2 * f()^{-1}$ ) |
| $f() * d_1 = d_2$   | divide by $d_1$ ==> | $f() = d_2 / d_1$    | ( $d_2$ dominates)            |
| $d_1 / f() = d_2$   | multiply by f() ==> | $d_1 = f() * d_2$    | (NOT $d_1 = d_2 * f()$ )      |
| $d_1 / f() = d_2$   | divide by $d_1$ ==> | $1/f() = d_2 / d_1$  | ( $d_2$ dominates)            |

These algebraic rules show that when there is a conflict the term that already dominates on one side still dominates.<sup>5</sup>

## 2.9 States and sequences

Besides domains, subjects and attributes, value structures also have states meta-data. States tell the time period that the value structure's primitive value(s) describe its subject's attribute. For example, during most of its current existence Halley's comet's mass is relatively stable. However, when it is close to the sun it loses mass dramatically as it heats up and forms a characteristic tail. Thus, while over the period of a typical day far from the Sun its change in mass is very small, over the period of a day close to the Sun its mass change is larger. What the value is depends on when and for how long you

---

5 At least two problems exist. One is that doing an operation, then undoing it, leads to a different result:

$$a + b = c \quad (\text{subtract dominant } a) \implies b = c - a \quad (\text{add non-dominant } a) \implies b + a = c$$

The hope, however (and this is just a hope) is that doing an operation and undoing it will not make sense under most algebraic circumstances, so that this contradiction will rarely manifest itself.

Another problem concerns division. When transforming as below:

$$d_1 / f() = d_2 \quad \text{multiply by } f() \implies d_1 = f() * d_2$$

we ought to remember that because  $f()$  started as a divisor, it's value cannot be 0. That is currently unstateable for an arbitrary expression. (One option might be to record this in the context.)

look.

Contexts and states exist to specify the time, time duration and other information on the applicability of a particular object's attribute value. States allow us to associate different values of one attribute with different times. They are grouped by how long they last. The single-valued attribute `statesFinitenessAttr` maps from a state to a member of `finitenessClass` (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 `SciRep` applies mathematical methods that consider infinitesimally small changes, like integrating functions. By convention their only useful attributes are `statesFinitenessAttr` and `statesTimeAttr` 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 may be queried with the `statesPrevStateAttr` and `statesNextStateAttr` attributes respectively. Finite states may be subdivided along two axes: timed vs. untimed states, and series vs. non-series states.
  - a. **Timed states** are given a specific time and duration. Their single-valued time attributes are: `statesBeginTimeAttr`, `statesEndTimeAttr`, `statesTimeAttr` (the midpoint time) and `statesDurationAttr`. Timed states also may have the attribute `statesScaleAttr` which maps from a state to a scale which tells information common to many states (like the duration). **Untimed states** are not given a specific beginning time, ending time or duration.

The following relationships hold among `statesBeginTimeAttr`, `statesEndTimeAttr`, `statesTimeAttr` and `statesDurationAttr` for a timed stated named `State`:

$$(\text{State} \rightarrow \text{statesEndTimeAttr} - \text{State} \rightarrow \text{statesBeginTimeAttr}) \\ = \text{State}.\text{statesDurationAttr}$$

$$(\text{State} \rightarrow \text{statesBeginTimeAttr} + \text{State} \rightarrow \text{statesEndTimeAttr}) * 0.5 \\ = \text{State}.\text{statesTimeAttr}$$

- b. **Series states** exist in an array of states where each is constrained to come directly after the previous state in the series. This ordering is created automatically when you give an array of states. Where a state is in a series may be queried with the `statesSeriesIndexAttr` attribute which start at 0 for the first. If they are *timed* then they must share the same scale and therefore have the same duration. **Non-series states** may also have `statesPrevStateAttr` and `statesNextStateAttr` attributes but because they are not defined in an array they must be defined specifically for each state.

Whether or not states have time information and whether or not they exist are two independent issues as is shown by the examples below:

- a. **Timed, series, finite state:** The United States conducts a census every 10 years. The time duration is 10 years making it timed and regular enough for the information to be placed in a series.
- b. **Timed, non-series, finite state:** The geological eras `paleozoic` (from 570 Myr to 245 Myr)

mesozoic (from 245 Myr to 66 Myr) and cenozoic (from 66 Myr to now) have time and ordering information, but their durations are unequal.

- c. **Untimed, series, finite states:** A population genetics experiment might iterate through temporally-distinct breeding populations of some initial population. The fact that they are in a definite series is important (the first population, the second population, *etc.*) How long each population lasts may be irrelevant.
  - d. **Untimed, non-series, finite states:** A ball hits a wall and bounces back. There is ordering information, but the time information is missing and the states are not “successive generations” of the same fundamental thing.
3. An **infinite state**, as its name suggests, lasts forever. It covers all time: all of the past, the present, and all of the future. The infinite state `universalState` is pre-defined. The state `universalState` is the *default* state of values and queries. If no state is explicitly given for a value or a query then its state will be `universalState`.

A sequence is a series of related states. Any finite state – whether timed, untimed, series or non-series – may be decomposed into a series of finer-grained finite states. The multi-valued attribute `statesStateDecompositionAttr` maps from a state to a state decomposition.

State decompositions may refer to the components of a state by either stating giving a class that all component states are in or by giving the starting and ending value of some attribute.

The state decomposition attribute `stateDecompositionsDomainAttr` maps to a class whose instances are the values of the decomposition. For example, geologists define three periods of the Mesozoic era: the Triassic, the Jurassic and the Cretaceous.

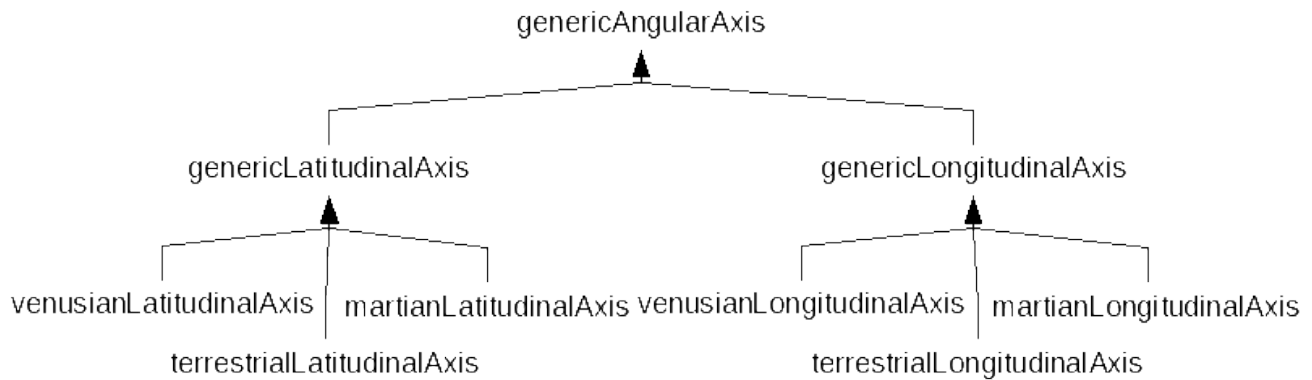
There are two forms of state decompositions. The first form has single-valued attribute `stateDecompositionsClassAttr` that maps to a class that is a subclass of `stateClass` and whose in-order listed instances are the states of the decomposition. The second form has single-valued attributes `stateDecompositionsStartStateAttr`, and `stateDecompositionsAdvanceAttrAttr`. The first attribute maps from state decompositions to the first state in the decomposition. The second attribute maps from state decompositions to an attribute that itself maps from states to the next state in the given decomposition.

The attribute `stateDecompositionsNumStatesAttr` maps from state decompositions to a non-negative integer that tells how many states are in the decomposition. This attribute is defined for both “set” and “linked-list” decompositions, and the code for it is analytical.

## 2.10 Axes

Axes allow values of the same dimension to be distinguished by the orientation and alignment in some potentially multi-dimensional space. For example, the axes `terrestrialLatitudeAxis` and `terrestrialLongitudeAxis` both have dimension `angle` but represent orthogonal axes.

Axis instances of the same dimension are arranged in their own **compatibility hierarchy**. Compatibility hierarchies allow more generic, ancestral axes to match with more specific, offspring axes if there exists a direct path from ancestor to offspring that does not have to go back up the hierarchy from the ancestor. For example, consider the partially compatibility hierarchy for axes of dimension `angle` given below.



It cannot hurt to reiterate that this hierarchy is not a class/instance object hierarchy (all objects listed above are instances of `axisSet`, none are subclasses), but an entirely new hierarchy of compatibility among instances. Axis `terrestrialLatitudinalAxis` is compatible with `genericLatitudinalAxis` and `genericAngularAxis`, but not with (for example) `venusianLatitudinalAxis` or `terrestrialLongitudinalAxis`.

Axes may have preferred units. For example, the preferred units of `genericAngularAxis` are radians, while those of `genericLatitudinalAxis` and its offspring, and of `genericLongitudinalAxis` and its offspring are degrees. Attribute `axesPreferredUnitsAttr` maps from the axis to its preferred units.

Before two values with different axes are compared, added, or otherwise treated by non-assignment binary operators they must be **aligned**. Axes `a1` and `a2` are already aligned if:

1. `a1` and `a2` are equal, or
2. `a1` is a direct ancestor of `a2` in the axial compatibility hierarchy they share, or
3. `a2` is a direct ancestor of `a1` in the axial compatibility hierarchy they share

However, if none of these relationships hold between `a1` and `a2` then the Scienceomatic looks for an explicitly-stated additive offset between the two axes. The attribute `axesAxisOffsetInfoAttr` maps from an axis to an instance of class `axesAxisOffsetInfoSet`. Fully described instances of this set have five attributes:

- `axesAxisOffsetInfosFromAxisAttr`. Maps to the axis that has the value to convert.
- `axesAxisOffsetInfosToAxisAttr`. Maps to the axis for which a converted value is desired.
- `axesAxisOffsetInfosOffsetAttr`. Maps to a value telling how much to add to the `axesAxisOffsetInfosFromAxisAttr` value to generate the `axesAxisOffsetInfosToAxisAttr` value.
- `areAxisInSameDirectionAttr`. Maps to either `true` or `false` depending on whether the axes are parallel or anti-parallel. The dimension `time` provides an example of anti-parallel axes. Most ways of talking about time have larger values corresponding to later times. Historians and archaeologists, however, often use switch axes to keep values positive. For example, instead of saying “year -500 of the Common Era” it is common to say “year 500 *Before* the Common Era”. For axis `beforeCommonEra` larger numbers correspond to *earlier* times. Please note: other than flipping directions there is no other way to “scale” one axis relative to another; *unit conversion* is used for scaling values of the same dimension.
- `canAxisBeSelfOrthogonalAttr`. Maps to either `true` or `false` depending on whether the axis is either can possibly be orthogonal with itself or not. For example, a generic

length axis can be orthogonal with itself (there are *three* spatial dimensions) but a very specific length dimension, like one that measures height above mean sea level, cannot be orthogonal with itself. There is only one sea level surface with which to be normal, and this permits only one axis. (Of course one may define other axes parallel or anti-parallel to the sea level axis with different origins, but no axis in this family can be orthogonal to themselves or each other.) This attribute is mainly useful for grouping axes together in coordinate systems. (Please see *Coordinate Systems*.)

For example, based upon 23 geologically stable tide gauges it is believed that mean sea level rose between 10 to 20 cm from about 1900 to 2000<sup>6</sup>. The notion “mean sea level” is then meaningless without some stated or implied time range.

We could handle this by *defining* the following relationship:

$$(\text{meanSeaLevel2000CEAxis value}) = (\text{meanSeaLevel1900CEAxis value}) + -15 \text{ cm}$$

This, of course, says that one must add -15 centimeters to a value with axis mean sea level as measured in 1900 of the Common Era to get the equivalent value for an axis with the mean sea level as measured in 2000 of the Common Era. We can represent this by encoding the following relationships:

```
meanSeaLevel2000CEAxis.axesAxisOffsetInfoAttr
  := meanSeaLevel2000CETo1900CEAxesAxisOffsetInfo
meanSeaLevel2000CETo1900CEAxesAxisOffsetInfo.
  axesAxisOffsetInfosFromAxisAttr := meanSeaLevel2000CEAxis
meanSeaLevel2000CETo1900CEAxesAxisOffsetInfo.
  axesAxisOffsetInfosToAxisAttr := meanSeaLevel1900CEAxis
meanSeaLevel2000CETo1900CEAxesAxisOffsetInfo.
  axesAxisOffsetInfosOffsetAttr := -15 centimeters
```

By the way, whether or not the rise was truly 15.000,000,000,000 centimeters is besides the point as this is a *mutual definition*. Later, if we get data that we believe to be more precise, we may use it to define another axis, say meanSeaLevel1900CERecalc2009Axis:

$$(\text{meanSeaLevel2000CEAxis value}) = (\text{meanSeaLevel1900CERecalc2009Axis value}) + -16.783 \text{ cm}$$

The Scienceomatic is successful at aligning two axes  $a_1$  and  $a_2$  if it can find some sequence of axes:  $a_1, a_{i_2}, a_{i_3}, a_{i_4}, \dots, a_{i_{n-1}}, a_2$  where, for all successive pairs of axes, either both members of the pair are already aligned (as detailed above) or an axesAxisOffsetInfoSet instance exists to convert from  $a_{i_i}$  to  $a_{i_{i+1}}$ .

Besides the axes listed above within the genericAngularAxis compatibility hierarchy, here are some more axes that are defined within std\_2009A\_phillipsj.culture.model (and there may be others):

genericCountAxis. Of dimension count (integer). Corresponds to a count of a number of “widgets”: distinguishable, countable things. Offspring of this generic axis count more specific

<sup>6</sup> Please see <http://www.globalwarmingart.com/wiki> and Bruce C. Douglas (1997). "Global Sea Rise: A Redetermination". *Surveys in Geophysics* **18**: 279-292. (<http://www.springerlink.com/content/p364381652174757/>)

things (for example, instances of `timePassageSet`).

`genericFractionAxis`. Of dimension `dimensionless (real)`. Corresponds to a fraction [0..1] telling the relative size of a part relative to a whole as a ratio.

`genericTimeAxis`. Of dimension `time (real)`. Corresponds to a time, for example, the time of an occurrence. Offspring of this generic axis correspond to specific origins and scales. They include `geologicalTimeAxis` (with preferred units `millionsOfYearsAgo`), `gregorianCalendarTimeAxis` (with preferred units `years`), `universalTimeCoordinatedAxis` (with preferred units `seconds`).

`genericLengthAxis`. Of dimension `length (real)`. Corresponds to a distance. Offspring of this generic axis include `meanHeightAboveSeaLevelAxis` (with preferred units `meters` and where the sea level during \_\_\_ is taken to be the standard) and `solarSystemDistanceAxis` (with preferred units `astronomicalUnits`), and `interstellarDistanceAxis` (with preferred units `lightYears`).

`genericMassAxis`. Of dimension `mass (real)`. Corresponds to mass.

`genericChargeAxis`. Of dimension `charge (real)`. Corresponds to electric charge.

`genericQuantumSpinAxis`. Of dimension `quantumSpin (rational)`. Corresponds to quantum spin.

`genericAreaAxis`. Of dimension `area (real)`. Corresponds to area.

`genericVolumeAxis`. Of dimension `volume (real)`. Corresponds to volume.

`genericVelocityAxis`. Of dimension `velocity (real)`. Corresponds to velocity.

`genericAccelerationAxis`. Of dimension `acceleration (real)`. Corresponds to acceleration.

`genericDensityAxis`. Of dimension `density (real)`. Corresponds to density.

`genericForceAxis`. Of dimension `force (real)`. Corresponds to force.

`genericEnergyAxis`. Of dimension `energy (real)`. Corresponds to energy.

`genericPressureAxis`. Of dimension `pressure (real)`. Corresponds to pressure.

`genericFlowAxis`. Of dimension `flow (real)`. Corresponds to an absolute flow of the number of countable things (“widgets”) per time, but per nothing else.

`genericFluxAxis`. Of dimension `flux (real)`. Corresponds to the flow of some thing through some area during some time. In other words it is both per area and per time.

An important usage of axes is to compose coordinate systems, which are discussed in later with `Coordinate Systems`.

### **3. Objects and their Manipulation**

Something here.

## 4. Calling the `Scienceomatic` from Java

Imperative flow control (`if`-conditionals, `for`-loops, *etc.*) was consciously excluded from `SciRep`. The reason for this is to strip the language down to what it really needs. Java, on which the `Scienceomatic` environment depends, already *has* all that. There is no need to needlessly replicate functionality.

What is needed is an ability to Java to access `Scienceomatic` functionality. The `Scienceomatic` provides the following:

## 5. Coordinate Systems and Data Display

Something here.

## 6. Explanatory Traces and Explanatory Stories

The Scienceomatic can be used to compute answers to scientific questions, but it was designed to compute *explanations* to scientific questions. Explanations are computed at two levels. **Justification traces** outline the reasoning used along one train of thought to compute one entity's attribute. **Justification stories** outline the reasoning used to partially order a set of justification traces (or other justification stories) in a partial order according to believability according to some criteria.

### 6.1 Justification Traces

Justification traces explain one way of computing a value or expression by telling all of a train of steps used to compute it. They always start with either the subject of the query, or an assertion from the context of the query. From this start they proceed by listing transformations from the previous step to calculate the next intermediate value.

These subsequent steps have form:

(<operation[*i*]> <optionalResult> <operationArg[1]> .. <operationArg[*m*]> )

where:

<operation[*i*]> tells the operation that was done on the value computed from step (*i*-1) to generate the next value.

<optionalResult> optionally tells the result of applying <operation[*i*]>. This value *must* be given if this is the last step in the train of steps. In this case it tells the final value or expression computed by the explanation. However, if step <operation[*i*]> is not the last step in the calculation then it may be replaced by nil.

<operationArg[1]> .. <operationArg[*m*]> tell the arguments to <operation[*i*]> to compute <optionalResult>.

In principle the operations used in justification traces may be from any type of reasoning, subject to the constraints of proper naming and non-ambiguity. The Scienceomatic architecture however inherently recognizes the following operations:

(context-process <process>). The step prior to this step gave a context. This step tells a process instance <process> listed in the given context.

(context-stasis <stasis>). The step prior to this step gave a context. This step tells a stasis instance <stasis> listed in the given context.

(object-class <class>). The step prior to this step gave an object. This step tells a class <class> to which the given object belongs.

(object-assertion <assertion>). The step prior to this step gave an object. This step tells an assertion <assertion> of the given object.

(assertion-expression <expression>). The step prior to this step gave an assertion. This step tells the expression <expression> of the given assertion.

(expression-solve <solvedExpression> <subjectAttributePairing>). The step prior to this step gave an expression. This step solves the given expression for the subject and attribute given in <subjectAttributePairing>. The result of this is a new expression given by <solvedExpression>.

(`expression-simplifyForComputation` `<simplifiedExpression>`). The step prior to this step gave an expression. This step generates an expression `<simplifiedExpression>` (perhaps which is the same one as was given) that is equivalent to the given expression, except that it may have been modified to be less computationally intensive to compute.

(`expression-compute` `<result>` `<expVal[1]>` .. `<expVal[n]>`). The step prior to this step gave an expression. This step computes a result of this expression `<result>`. To do so it perhaps uses the values of subexpressions given in `<expVal[1]>` .. `<expVal[n]>`. Each value of subexpression expression has format:

(`<subexpression>` `<value>` `<optionalJustificationTrace>`)

where subexpression `<subexpression>` is listed as being computed to value `<value>`. If it is given and is non-nil then `<optionalJustificationTrace>` tells a justification trace for the computation of `<value>` for `<subexpression>`.

The default operation of the `Scienceomatic` architecture is to list the `<optionalResult>` for trace steps that are not the final one only if there are multiple possible values. An example of a reasoning step with multiple possible values is `context-process` because multiple processes may be listed as active in the context. In this case the particular process `<process>` would definitely be listed. However, an example of a step with only one resulting value is `assertion-expression`. Each assertion has precisely *one* expression, so by default the `Scienceomatic` architecture does not bother listing it. To force the `Scienceomatic` to list all values in generated expressions \_\_\_\_.

## **7. Managing the Knowledge and Defining New Cultures**

Something here.

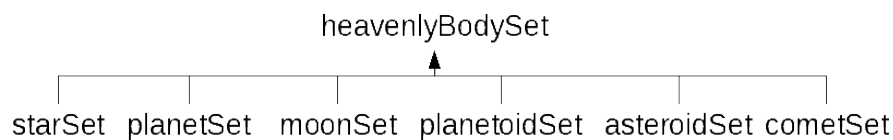
## 8. Managing the Ontology and Being Scientifically Conservative

The organization of ontology matters in terms of (1) how efficiently the knowledge base computes, (2) how large the knowledge base is, and (3) how easy it is to understand the knowledge base. In this chapter we outline several principles for keeping the knowledge base organized and efficient.

### 8.1 Organize sets of objects by their intrinsic properties, not by what they happen to be doing at the time

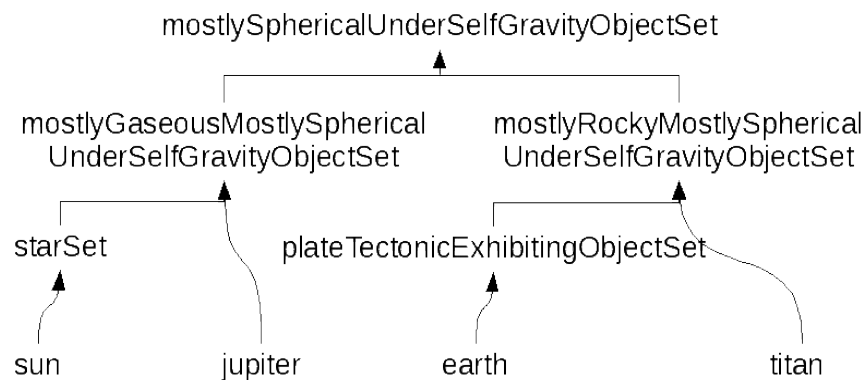
Sets should group things with the same inherent properties, not with the same happenstance. We expect happenstance to change more frequently than inherent properties.

For example, our solar system is commonly described as being composed of the Sun, some number of planets, the moons of those planets, and numerous planetoids, comets, asteroids, *etc.* One might be tempted to organize this in the following ontology:



This organization has the advantage of mirroring how we categorize items by size, the object that they orbit, and somewhat by composition. However, the object that they orbit is mostly arbitrary. Saturn's moon Titan is as large as Mercury, and hence would be considered a planet if it independently orbited the Sun. Further, consider Pluto. Initially considered a planet, then considered a planetoid, Pluto was hypothesized to initially be a moon of Neptune before tidal forces spun it into an independent orbit around the Sun.

A better ontology organization uses size in a grosser fashion, composition more explicitly, and concentrates on the behavior exhibited by each body internally and independently of what it orbits and what orbits it.



Here *is-a* relationships are represented with rectilinear lines and *instance-of* relationships are represented by curved lines. Rather than make an explicit size distinction between planet, planetoid, and even moon and star, the improved ontology uses makes just one size distinction: between having enough gravity to pull itself into a roughly spherical shape, or not. The advantage of this is that all members of class `mostlySphericalUnderSelfGravityObjectSet` can inherit attribute `shapeAttr` value `sphere`.

Considering just those items with enough gravity to be roughly spherical, the next split is on composition. The rocky dwarf planets and many of the moons would be under mostly rocky, while the Sun and the gas giant planets would be under mostly gaseous. Under mostly rocky the distinction could be made between those exhibiting plate tectonics (for example), and those that do not. Under mostly gaseous the further distinction can be made between those that have gravitational-mass self-sustaining thermonuclear reactions (“stars”), and those that do not.

## ***8.2 The difference between sets and collections***

There is a difference between a collection and a set. A

## **9. Managing Trajectories of Scientific Belief Over Time and other Philosophical Considerations**

Something here.

## 10. Managing

Something here.

## **Appendix I: The domains of std\_2009A\_PhillipsJ:**

Something here.