Contents

Introduction to semantic meta-modeling with Thinklab: a user's guide.

Modeling is the production of simulated observations of a system, meant to help understand its structure or behavior. Armed with the power of creating such observations, we can visualize and analyze the results of choices we could make in real life before we actually make them. Modelling practice has advanced to a point where very complex models can be described succinctly using sophisticated mathematics. This sophistication, however, usually applies to the mathematical instruments used but does not extend to the semantics (conceptual description) of the systems under investigation. The conceptual view of the system incorporated in a model is usually very basic: time may be a number starting from zero, initial conditions are three-letter "variable" identifiers hidden in configuration files that change completely for each model, etc. The lack of a solid, systematic and self-consistent conceptual description of the system modeled stands in the way of model integration, internal consistency checking, communication and validation of model structure, and ability to reuse models in different contexts or together with others. All this diminishes the value of existing models to the point that they can often only be used effectively by their developers. Unless you have written a particular model (and you have done it yesterday) extending or upgrading it is often difficult, time-consuming and fraught with doubt about what the various model components actually mean.

This documentation is an initial attempt to describe semantic meta-modeling, a methodology that uses explicit semantics and machine reasoning to support integrated, multi-paradigm, multi-scale model development. The Thinklab language described here is the instrument we are developing to make this vision practical. Please note that everything about this notes (even the Thinklab name) is in its initial stages and should be considered both confidential and subject to change.

> *The language, software and documentation have been developed by Ferdinando Villa with the collaboration of the ARIES project team, including but not limited to Kenneth Bagstad, Brian Voigt, Gary Johnson, Ioannis Athanasiadis, and Luke Scott. Please consult with the first author at ferdinando.villa@bc3research.org before sharing or making any use of this material unless previously coordinated.*

Semantic meta-modelling

Thinklab aims to address the task of "integrated modeling", which reconciles strong semantics with modeling practice, helping achieve advantages (such as modularity, flexibility, validation, and integration of multiple paradigms and multiple scales) that have remained largely unrealized to this day. To achieve this goal, Thinklab keeps the logical representation of the modeled world distinct from the algorithmic knowledge that allows it to be simulated, and uses artificial intelligence to assemble the latter into the best possible algorithms to simulate a system described by users in a purely conceptual and much simpler way.

As suggested above, semantic modeling, or more accurately semantic meta-modeling, uses the idea of **observation** as the unifying theme to define a general way to model physical objects and phenomena (Fox and Hendler 2009). A model is a strategy to produce observations of a concept that comes from a shared knowledge base. **In order for models to be compatible and be capable of being used as component of the same computation, it is sufficient that the abstract knowledge they represent is compatible.** Modern artificial intelligence provides algorithms and tools to automatically validate the consistency of an abstract knowledge base. This way, the approach enables the integration of many modeling paradigms that are often applied separately, for example spatially-explicit to process- and agent-based models, or probabilistic and deterministic models. This conceptualization builds a natural path to reach goals in modeling that have frequently been discussed, but not demonstrated to their full potential, including modular modeling, multiple-paradigm modeling, multiple-scale modeling and structurally variable modeling.

The logical representation is modeled using concepts and relationships that comprise Thinklab's abstract knowledge base, built out of ontologies (more on this later). In the abstract knowledge base, concepts such as "watershed," "elevation," or "temperature" are defined, along with information on how they may relate to each other. No attempt is made to indicate how their models may be computed.

The algorithmic knowledge base is where you can provide models so that simulated "observations" can be computed. At the user's request, the artificial intelligence in Thinklab will choose algorithms from this knowledge base and build an integrated algorithm by assembling them, driven by the abstract semantics. The result of calibrating and running the integrated model is the production of observations of the concepts contained in the abstract knowledge base.

A model in Thinklab represents a strategy to observe a concept. Models can consist of entire simulations, simpler algorithms, datasets, or even simple numbers; from the Thinklab point of view, all these just represent different ways to observe a concept. In striking difference from almost all mainstream modeling approaches, numbers, data or equations have absolutely no meaning by themselves, even with descriptive names or associated with formal metadata: even the simplest number can only be used in Thinklab if it has a concept associated with it. Thinklab forces you to use concepts so that "metadata" in your models only document auxiliary information as they should; the conceptual part of the knowledge base serves automatically as the documentation of the models, while at the same time providing a layer of "meaning" on which collaboration and model integration are based.

Structure of this document

This documentation contains, for now, five main chapters and a glossary, subdivided as follows:

1. Chapter 1, Collaborative infrastructure and the semantic modeling workflow provides generalities about both the Thinklab language and the collaborative environment it is expected to support. Initial examples of semantic model specification are illustrated.
2. Chapter 2, Models as observations: subjects, qualities and traits introduces the semantic framework for observable concepts that we adopt in Thinklab.

3. Chapter 3, Connecting data to models: semantic annotation and observation semantics introduces the semantics of observations that can be connected to observables, and the language statements that implement it. The examples concentrate on the semantic annotation of existing information such as data.

4. Chapter 4, Computing deterministic and probabilistic observations extends the material of Chapter 3 to the case of computed information, which in Thinklab is uniform with data, and to providing semantics for externally computed information.

5. Chapter 5, Making model choices depend on context, details the notion of scale in Thinklab, and the ways that specific model choices made by the reasoning algorithms are made and how they can be constrained to specific scales or otherwise influenced. In the process, conditional models, scenarios and the ways to influence Thinklab's resolution of models to concepts are described.

The chapters above provide a basic user guide to the new user of Thinklab, meant to support in-class instruction, and are by no means exhaustive of the system for either the conceptual side or the implementation. Not all examples in the chapter are guaranteed to work at all times; links to sections of this documentation that appear in the text may lead to non-existing pages. Any new documentation will appear in this section of as it is completed and reviewed.

The current documentation was written by Ferdinando Villa and reviewed by the collaborators listed above, with the addition of Stefano Balbi, Aiora Zabala Aizpuru, Elena Perez-Minana and Simon Willcock. The Bayesian network primer was written by Kenneth Bagstad, who also compiled the Glossary, with the collaboration of the authors listed. Gary Johnson wrote the cookbook examples. # Module 1. Collaborative infrastructure and the semantic modeling workflow.

Thinklab is a computer language and software toolset that supports a modeling paradigm where carefully designed components can be shared within a broad modeling community, without previous coordination. To achieve this, all model components are semantically annotated, i.e., associated to generally recognized concepts, such as "watershed" or "elevation." The operation of annotation establishes, explicitly and unambiguously, the identity of each modeled entity and its boundaries of validity for it (e.g., a particular spatial region or temporal period). Using a shared set of ontologies (computer documents that define concepts and their relationships) ensures that independently developed model components can be linked without ambiguity or error and be used by a growing global research community.

The components needed to achieve this vision are:

1. A web-based infrastructure that connects model developers, helps coordinate knowledge and model sharing, and controls data access and distribution in a way that correctly supports collaboration.

2. A suite of shared ontologies to ensure a stable common language for data and models, modified only through a carefully designed, collaborative workflow that facilitates conflict-free ontology extension.

3. A modeling language and server (provisionally named **Thinklab**), and an end-user software toolkit (**Thinkcap**), which enable model development with semantic annotations based on the shared ontologies. Thinkcap is a graphical user interface (GUI)-based client that communicates with one or more Thinklab servers to publish, run, and share models and knowledge. The Artificial Intelligence for Ecosystem Services (ARIES) project uses Thinklab for its underlying operation; Thinkcap is the primary means of model development for ARIES modelers. Other applications of Thinklab and Thinkcap are possible outside the field of ecosystem services.

This document is the first of a suite of modules aimed to serve as reference and guide for a brief course in semantic modeling. As this material is fairly novel, do not worry if it is not totally clear after the first read. The text below provides a brief introduction to the three key components listed above.

Component 1: the collaborative, web-based infrastructure

A complete semantic modeling platform can be installed on a desktop computer and used by a single modeler. However, the full value of this paradigm can only be achieved when the models and the knowledge they represent are shared with a broad model developer community, so that each researcher can concentrate on, and contribute scientific advances for, the issues that reflect their expertise and interests. The semantic modeling approach facilitates integration across all modeling disciplines through the use of common concepts.

The collaborative infrastructure that supports this sharing is a web-based application that allows the desktop application to synchronize with a distributed set of common ontologies, data, and models. The synchronization is automatic using the Thinklab/Thinkcap software suite, and guarantees that the "core" concepts used by all model developers are the same. Model developers can register to obtain official certification and join specific modeling groups, which give access to different "views" of the shared knowledge base. Each user in the same group will use the same core ontologies, and the collaborative infrastructure grants access to public models built by all model developers that belong to the same groups. Groups can be hierarchically organized to fine-tune access to restricted datasets without compromising access to core and public knowledge.

Use of the software requires registration at [integratedmodelling.org/dashboard] and a request to the site administrator for certification. Certification results in a certificate file being emailed to the user, which is copied in the `.thinklab/` directory under the user's home directory and is automatically loaded by the software to identify the user and allow access to all shared resources for the groups s/he belongs to.

Component 2: Semantic modeling and ontologies

The central tenet of semantic modeling is to maintain the meaning of anything being modeled. This emphasis on meaning facilitates model sharing with other, independent modelers: if the meaning of concepts is consistent throughout the modeling community (including, and not limited to, their temporal and spatial properties, and all relationships between them

and other concepts), the models developed for those concepts are automatically compatible. The Thinklab system is specifically designed to take advantage of this semantic modeling approach. Concepts are used to describe each entity in a model using a programming language whose syntax is optimized for this task.

The semantic modeling workflow differs from that of other, more common, modeling approaches. Specifically, model development requires first of all that a specific subject is identified that is described by a concept representing thecontext for all subsequent modeling (e.g., a watershed). This concept is crucial: describing an area as a “region” has a different meaning than describing that same area as a “watershed”, as models that work on a watershed (for example, to extract the stream channel network) will not work on a region. Additional concepts can be identified to describe each quality (e.g., elevation, slope), process (e.g. erosion, rainfall), subject (e.g. household, city, river) orevent (e.g. inundation, fire) of interest.

Component 3: The Thinklab modeling language and Thinkcap software environment

The Thinklab modeler workflow uses the Thinklab modeling language to create models for specific concepts and define their context of validity. For most users, model development and testing will take place within the Thinkcap software environment., which provides an intelligent editor for the language and facilities to interact with servers to synchronize and search knowledge, test, run and publish models. Thinklab also supports a user workflow, which we will not discuss in these notes aimed to modelers: this workflow is supported by a separate web application (in course of development at the time of this writing) and is of much simpler nature, aimed to allow non-technical users to search for concepts, define a context and run models using simpler metaphors to use the library of knowledge created by modelers without being exposed to model details.

Any model, with the exception of data (NOTE: in Thinklab, data are models) may require the observation of further concepts, besides its own observable, in order to be computed. For example, a model of vegetation growth may require rainfall, soil and temperature observations. In Thinklab, as mentioned, all these dependencies are expressed using concepts; Thinklab applies search algorithms to iteratively match concepts to models as many times as necessary, resolving concepts to models as new dependencies are brought in by the models selected. Heuristics and artificial intelligence are employed to define the most suitable model at every step, based on both objective and subjective criteria (such as modeler-defined data quality). This process, called resolution, enables Thinklab to integrate a large and distributed model base and make it accessible to non-technical users: since queries are performed onconcepts, running any model is paramount to conducting a web search for the concept(s) of interest. Indeed, the combination of a shared model base and the Thinklab language and infrastructure can be thought of as a semantic webimplementation specialized to handle and serve “live” model knowledge.

A modeling workflow example

Creating a context

As mentioned above, running a model requires establishing a context first of all. A context can represent any kind of subject, but in our example we will use mostly contexts that are associated to spatial locations, the most useful for the type of ecoinformatics modeling we use in applications. So we will talk about context regions, watersheds, villages and the like. The spatial aspect of a subject pertains to its scale, as one of the extents in it; those may also include time and potentially others, which will not be discussed here. A geographic extent can be defined using Well-Known Text (WKT) : see the glossary for a definition and example. A simple geographical polygon using WKT can be written by listing a projection code followed by sets of spatial coordinates, like in:

```
EPSG:4326 POLYGON((-70.8783850983603
-3.3045881369117316,-69.05465462961
-3.2661991868835875,-69.13155892648547
-4.575963434877864,-70.91683724679801
-4.630717874946029,-71.15853646554768
-4.263784638927346,-70.8783850983603
-3.3045881369117316))"
```

The above WKT can be used within a simple Thinklab statement to create the Leticia region of Peru, using the subject type`im.geography:Region` (a concept from the`im.geography` ontology that defines a simple region of geographical space – see below). The full statement is written as:

```
observe im.geography:Region leticia-peru over space(
      grid = "500 m",
      shape = "EPSG:4326 POLYGON((-70.8783850983603
-3.3045881369117316,-69.05465462961
-3.2661991868835875,-69.13155892648547
-4.575963434877864,-70.91683724679801
-4.630717874946029,-71.15853646554768
-4.263784638927346,-70.8783850983603
-3.3045881369117316))"
);
```

The observe statement instructs the system to create a region named ‘leticia-peru’, whose spatial scale will be represented by a 500-m cell size over the polygon defined by the WKT. A couple things to note. First of all, the identifier for a concept in the Thinklab language is composed of two parts separated by a colon. The first is the namespace, which corresponds to a single file that contains concepts and models for a particular topic (e.g., im.geography). The second is the name of the concept within that namespace (i.e. Region). Simply stated, the core concept Region, a generalized geographic region, is found in the im.geography namespace. (Naming conventions and a capitalization scheme have been adopted and will be explained in detail later).

Another important point. It is easy to get confused by thinking of the “polygon” as the “context”. Indeed, a context is the whole subject created by the statement, and its identity is given by the `im.geography:Region` concept – theobservable for this subject. We have said above that the whole purpose of semantic modeling is to associate models to concepts. Indeed, when the above statement is “run” to create the subject and make it the context of further modeling, the resolution algorithms in Thinklab search for models associated to the observable `im.geography:Region`. Since no special semantics are attached to regions, no models are (by default) associated to this concept, as a region is sufficiently specified by assigning its spatial (and possibly temporal) scale. If, however, we had intended our region of interest to be a watershed, we could have written instead:

```
observe im.hydrology:Watershed leticia-peru over
space(
      grid = "500 m",
      shape = …
);
```

While the statement looks almost exactly the same, the meaning is very different, as we are now defining a subject with much more conceptual detail associated. If nothing special besides the scale is needed to characterize a semantically consistent `im.geography:Region`, a `im.hydrology:Watershed` has important restrictions: for example, the watershed has a shape that excludes areas that are not hydrologically reachable by rainwater. For all practical purposes, a watershed is a region, but it has additional semantics: using the knowledge visualization facilities in Thinkcap, you can explore the im.hydrology ontology to reveal that `im.hydrology:Watershed` is indeed linked to `im.geography:Region` by an is-a relationship, meaning that a watershed is a special region (but a region is not a special watershed).

The result of this tiny change in the specification is quite dramatic. When the watershed above is used as a context for modeling, Thinklab will check the ontology (`im.hydrology`) and find that some functional properties are required in order for a watershed to be semantically consistent. For example, all hydrologic observations on a watershed require that a digital elevation model, flow direction attributes and a stream network be defined. Faced with this need derived only by semantic analysis, Thinklab will look for a subject model that can produce these additional observations to complete the definition of a watershed when it is created. You can provide your own, custom-developed models for the same purpose, and tell Thinklab in which conditions it should be used (you can do the same also for a Region). As a default, the one model associated with `im.hydrology:Watershed` will run the set of hydrological analyses that comes with Thinklab, depending on being able to observe the elevation over the spatial scale defined for the watershed. If the elevation concept (`im.geography:Elevation`) can be resolved to a model in the knowledge base, this model is run to observe it, and the Watershed model is then run. The result is the computation of the basic hydrological characterization of the watershed, which includes the pit-filled elevation, the total contributing area for each point in the watershed, and the stream network in the region. All this happens without any user intervention: Thinklab simply recognizes that in order for a subject to properly represent a watershed, these quantities must be known, and proceeds to observe them based on the available knowledge base.

The modeling client software (Thinkcap) makes all these operations more intuitive by providing a workflow to create subjects and making observations in their context. Once the observe statement has been written, a marker named ‘leticia-peru’ will appear in the Navigator window of the Thinkcap interface. Dragging and dropping the ‘leticia-peru’ marker into a “context” window creates a subject, named leticia-peru. If the watershed version of the statement is used, this triggers the search for elevation and the computation of hydrological properties. Now that a subject has been created to use as a context, the interface will display the “root” subject named `leticia-peru`. The map window of the interface will display the area of interest. Dragging other concepts onto the map allows triggers their resolution to models; if an adequate model is found, the concepts are observed by computing the model, and visualized in the same interface.

Observing concepts in the context.

Once a subject has been created, you have a context in which you can observe more concepts if you want to. In the modeler software Thinkcap, you can locate a concept (from the Navigator or the Knowledge Search window) and drag it in to the context window; this is the equivalent of attempting to observe it in that context. As a response, Thinklab will evaluate whether the concept is able to be successfully resolved, i.e. a model associated with the concept can be found and run. If a valid model for a concept in that context is identified, it will be run and the results will constitute new observations of that concept. Other observations may be made during the computation, and those will also be available to visualize. The resulting observations will be displayed in the interface once a model has run to completion.

If the effort is unsuccessful, the model base doesn’t have a suitable model, but you can write one and assign it to the concept. If you are interested in observing a concept that you cannot find in the ontologies, both the abstract knowledge (the concept, contained within ontologies) and the model knowledge (models) will need to be defined. This documentation emphasizes model development. However, the complete software documentation contains details on creating abstract knowledge as well.

> *NOTE: Abstract knowledge plays a significant role in this modeling approach and affects usability at all levels. The creation of abstract knowledge should be approached carefully and collaboratively to ensure its utility extends beyond an individual modeling effort.*

Concepts may describe qualities, processes, events or subjects (there are also traits which occupy a special niche and will be discussed later). Observations of qualities correspond to what is commonly called data.

Qualities

Concepts that describe qualities (e.g., land cover type, temperature) define entities that cannot “stand alone” (note: we’re playing with semantic fire here, and we strive for consistency in using terms that are colloquial but have special meanings in our discussion. We use the word entity to mean anything, devoid of further significance. The topic does not leave the luxury of using terms like thing or object without raising doubts, so we will stick to that). Qualities mustinhere to a subject in order to be semantically consistent. For example, land cover type can only exist on some kind of “land” – so a geographical region (e.g., a country, a watershed) subject is necessary. Temperature can only be measured within a specific realm (e.g., the lower atmosphere, the ocean’s surface). As a result, an `observe` statement cannot be used with a quality concept where we put, for example, `im.geography:Region`. Thinklab will show an error marker and refuse to create your subject if an attempt is made to `observe` anything that doesn’t describe a subject capable of standing alone as an independent object.

Once a subject has been established, all concepts that are compatible with being its qualities can be observed. The universe of concepts is held within the core ontologies and can be added to using the Thinklab language. In Thinkcap, the Knowledge Search view allows to search for concepts by name; double-clicking a concept resulting from the search displays the knowledge graph, which graphically relates the selected concept to the full knowledge base.

Subjects, processes and events.

Subjects, processes and events are “identities” that can stand alone semantically: you can recognize a subject or event without having to refer to another subject as context. They can, of course, also be observed within the context of another subject: for example, observing the subject concept `im.infrastructure:Bridge` within a `im.geography:Region`, if successful, will create all the Bridge subjects in that region, using the scale of the region as a guide to find bridges that exist in that space and time. While we do not discuss much detail here, the difference between the different types above deserves some more explanation. These kinds of

concepts can be observed in the same way, but the results of observing a process or a subject are slightly different. While subjects have an identity that persists beyond their observation (i.e., observing them creates subjects that can be visualized independently and serve as individual contexts for other observations), processes and events are inherent to the subject they're observed in: an ecosystem service, for example, is a process inherent to a region that affects its subjects (ecological and social) and causes certain qualities (e.g. environmental values) to exist for them. The most important difference is that observing a process does not generate an independent entity for it, because processes occur within subjects. The same applies to events, which are really processes but are seen as atomic with respect to time, and therefore treated differently with respect to the scale of the root subject. When observing a process or an event within the context of a subject, any qualities or subjects produced by the process will be assigned to the context subject instead of the process itself.

Module 2. Models as observations: subjects, qualities and traits.

Anything that happens in Thinklab is the result of choosing a model and running it in order to create an observation. A model always stands between the intent to observe a concept and the successful observation of that concept. Models require a subject in order to observe anything, so a model begins with the observe keyword, which can be thought of as the declaration of the root subject. While most of the examples discussed in the text present the root subject as a region of space, this is only due to the nature of the examples. Space and time are special observation types, also known as extents; one or more extents create a scale. Special observation types will be discussed in greater detail in Module 5.

Because the observe statement does not specify how the subject should be observed, Thinklab will identify an appropriate model that will, if necessary, initiate any computation(s) required to create a semantically consistent subject. In the case of subjects, a model does not need to exist in the model base; if there is no model, a default set of actions will be taken, according to the concept that the subject incarnates. At a minimum, a simple subject will be created; this, as discussed in Module 1, is what will happen for example for a im.geography:Region, which is simply expected to be there and not required to have specific qualities. When, on the contrary, the semantics of the subject defines specific constraints, more observations may happen automatically. For example, observing a watershed subject would trigger a basic hydrological characterization of the watershed, consisting in observations of some qualities such as the flow direction and the elevation.

In contrast to subjects, models of a quality, must exist in the knowledge base: it is not possible to create a "default" elevation measurement or land cover classification. The simplest (and, from an epistemological point of view, also the best) model of a quality is evidence in the form of data. So when observing a quality, the model base is first searched for a data model (also called a data annotation) that matches that quality. If a data annotation does not exist for the quality, Thinklab will look up a model that can compute it based on an algorithm or other process.

Modeling in Thinklab entails writing model instructions that produce observations of subjects and qualities. This includes annotating data sources to turn them into models of shared, recognized concepts. An important part of this process is understanding the criteria with which Thinklab ranks models when more than one is found for a particular concept. These criteria, discussed in detail in Module 5, ensure that the most appropriate model will be selected to observe a concept in a specified context.

A `model` statement can be run manually in Thinkcap, by dragging and dropping it onto a context. While this is an appropriate way to test models, it is important to remember that the ultimate purpose of a model is to be considered during the resolution of the concept(s) it describes, so that Thinklab can choose the most suitable model for an observation in the specified context. In a collaborative effort, it is normal to have more than one model for a concept (e.g., different data sources with different resolution, coverage, currency). The criteria that negotiate the model selection process selected are the subject of Module 5.

Concepts and observables

At this point, it is important to understand the details of the concepts that are used to specify the observable of a model, i.e., the concept that the model will produce an observation of. Concepts live in ontologies and Thinklab can be used to define them (a Thinklab namespace is indeed an ontology, which can define semantics both for concepts and models). A substantial set of concepts come with the integratedmodelling.org certificate. The process of creating ontologies (or even new concepts in an existing ontology) has its difficulties. Creating new concepts or ontologies (as discussed in Module XXX), requires a deep understanding of the contents of existing ontologies and a cautious approach which ensures semantic consistency and avoids unintended consequences to the collaborative modeling environment. Ontologies are, the fundamental building blocks of the semantic modeling approach. The remainder of this section describes how to use the knowledge that exists within the ontology library.

Three fundamental aspects must be correctly specified to define an observable:

1. The **primary observable**: it is a thing, process, quality or event.
2. Any **traits** that further specify the primary observable and influence the type of observation made;
3. The **inherency** of a concept to a subject, which further informs the matching of concepts to models during resolution.

The primary observable

There are two fundamental kinds of observables:

1. Those that specify entities that can be thought of as existing autonomously (or, as some philosophers say, have unity): those include entities that "are", such as regions, human beings, animals or plants, but also entities that "happen", such as events (e.g. an earthquake) or processes (e.g. "carbon sequestration"). In the following, we use the convention of referring to all of these with the term **things**.

2. Those that specify **qualities**, which must refer to a thing in order to exist as a meaningful concept for modeling. Those include “properties” such as color, temperature, or density.

We say that an observable is concrete when it can be directly observed without further specification: for example, “nitrogen amount” would be concrete while “amount” would be abstract, the opposite of concrete. An “amount” cannot be identified without specifying an identity for it that “grounds” it to the physical world.

Because an identity clearly is also a concept but cannot be observed without an observable to refer it to, we need to bring in another class of concepts, which we call in general **traits**. Those are not observables: so you cannot write a model for, say, a plant species or a chemical element. Rather, they are used to qualify observables so that their observation becomes unambiguous. We recognize three types of traits:

1. **Identities**, which have the property of being able to turn a compatible abstract observable into concrete. An observable can have one and only one identity – for example, a “cat individual” or a “gold weight”. You cannot observe gold, but you can observe its weight: to annotate this observation, you use the abstract observable (weight, a quality) identified with “gold” (a chemical identity).

2. **Attributes** that can only apply to concrete observables and further specify them so that there is no ambiguity in annotations. For example “annual rainfall amount” would be observable without the attribute “annual”, but it would be incorrect to mix annual measurements with monthly ones, so we use the attribute to specify the annual character. There is a vast taxonomy of attributes, many of which are collected in our core ontologies so that the need for “inventing the wheel” is minimized (along with the risk of making the wheel square). An observable can have many attributes, but only one per category: so we can see “annual average rainfall amount” but not “annual monthly rainfall amount”.

3. **Realms** are special attributes that are very common in modeling, so we have decided to give them a status of their own. A realm refers to a broad subdivision of the physical world (not necessarily geographically delimited) where a particular observable is expected to be observed; for example, geographical realms such as land, ocean, atmosphere, soil, or biogeographical ones such as ecozones. They are conceptually not very different from attributes and they work the same way – many realms, but only one per category. Because modelers often use realms, we conceptualize them separately so they can be more easily catalogued and located.

Thinklab provides base semantics for several other kinds of observables, such as basic physical properties (see Module 3. Still, the distinction between things and qualities, complemented by traits, remains the fundamental conceptual skeleton for the process of annotation. Any confusion over what is a thing vs. what is a quality will lead to trouble. When possible Thinklab will validate concepts so it should not be allowable to use a thing concept when a quality concept is needed (or vice versa). This is not universally possible, and observations resulting in strange model behaviors will arise unless the distinction is clearly understood.

Observing things and qualities works differently, and the observation of each produces different outcomes.

- The observation of a thing, also known as a **direct observation**, produces a virtual representation of one or more things in a context. So for example, when observing a watershed, an “object” tagged with the watershed concept is created in the computer’s memory. Within the context of the watershed it is possible to observe households, rivers or bridges. The observation of those concepts will create new objects, under the “ownership” of the containing watershed.

- The observation of a quality is an **indirect observation**. An indirect observation produces an output (numbers, for example) that indirectly describes the state of that quality in the context. For example, consider the concept describing the amount of annual rainfall in a watershed. Let’s assume that the watershed context is distributed over an interpretation of space that conceptualizes it as multiple cells or polygons (In Thinklab, the temporal and spatial qualities of observations depend on the specific characteristics of the context. This will be discussed in greater detail in Module 5.) A successful observation of this concept creates an observation of rainfall in the watershed, most likely expressed in mm (expressing the density of the water volume over space). The observation will take the form of a map, attributing a number value to each subdivision of the context. Each value expresses the rainfall amount indirectly, referring to a known scale (how many mm, defined in the SI system).

The interpretation of numbers (or other data types, e.g., categories) in an indirect observation will depend on the defined observation semantics (for example, a measurement in a particular unit). Units and observation semantics are described in Module 3. For now, it is important to understand that the observation of a quality can only happen within the context of a thing. Temperature, a quality, cannot exist alone but only in reference to the thing whose temperature is being measured. Choosing semantics to match the desired level of detail enables semantic shortcuts with respect to the physical world. For example, it is possible to measure the “atmospheric temperature” of a “region” (a thing) instead of observing the “atmosphere” in that region and contextualizing the temperature to it.

Thinklab offers the user extensive granularity when defining concepts with its suite of existing ontologies. Things can be inanimate or reactive (agents); processes and events are specialized things that will be discussed in more detail later. For now, understand that most of what is referred to as “data” are qualities in the Thinklab language, and a “dataset” (a collection of different data relative to the same context) is what Thinklab considers the observation of thething that has been chosen as the context, including all of its observed qualities.

Keeping ontologies simple

We have discussed how traits work as “descriptors” to avoid confusion with other incompatible concept. For example, when modeling rainfall on an annual time scale, the results are presented as “annual rainfall” and annual measurements are not mixed with monthly or daily measurements. In a semantic world, it is also possible to represent such distinctions be represented by using a separate concept: for example `AnnualRainfall` and`MonthlyRainfall`, which could be specialized cases of `im.climate:Rainfall`. it is important to minimize the size of ontologies if they are to be used by wide communities, and because there are many attributes that apply equally to concepts for many disciplines, traits offer a way out of a potential semantic explosion that can lead to a complete lack of interoperability. If it were necessary to define a version of all the qualities that can be measured annually, the ontologies would rapidly increase in both size and complexity (e.g., monthly measurements for each month, daily measurements for each Julian date). Instead what is needed is the specialization of a concept that uses a general “adjective”, such as: -finite/infinite, -vulnerable/invulnerable, or -high/medium/low.

When looking up models, Thinklab prioritizes models that share the same trait as the concept that is being observed, and ignoring those models that feature a different trait of the same type. Traits can be added to concepts simply by writing them along with the main observable, as it would be done in English: for example adding the Annual descriptor to the context (`im.hydrology:Watershed`) in the observe statement:

```
observe im:Annual im.hydrology:Watershed
      over space(...)
model ... as measure im.climate:Rainfall in mm;
```

An “annual watershed” doesn’t make much sense by itself, but the Annual trait refers to the observation of the watershed, ensuring that any model or data selected to observe a quality in this watershed will be annual. So when observing rainfall in this watershed using the model statement, only observations of annual rainfall will be made. If a model is chosen to resolve a process concept, for example “river flow,” only models with an annual time step will be chosen. In other words, traits “percolate” through the resolution process and influence the choice of models made so that the result is always consistent with the observable concept.

Of course the observe statement above does not require any traits. They can be added to the model statement instead. The process will work the same way, and the resolution of an annual model will ensure that all observations are made on annual data (or processes with an annual time step). And, of course, it is critical to use appropriate traits when annotating data. For example, some traits detail the data reduction choices made when collecting data, such as average, minimum or maximum measurements. As many traits as necessary should be used: restrictions on the types and number of traits have been detailed before, but in general, only one identity is admitted, and as many attributes or realms can be present as long as each one belongs to a different abstract category. The following model statement illustrates how to assign traits for average annual rainfall data:

```
model .... as measure im:Annual im:Average
im.climate:Rainfall in mm;
```

Traits are also useful when classifying observations. They can be used to classify continuous data into discrete categories, which proves useful for certain types of modeling (e.g., Bayesian modeling). In this case, classifications can be done using the **by** keyword followed by the type of trait:

```
model Elevation as classify (measure
im.geography:Elevation in m) by im:Level into
      im:Low if 0 to 350,
      im:Medium if 350 to 1000,
      im:High if 1000 to 8000;
```

Additional examples of this will be shown in detail later. For now, consider the benefits of using the general trait im:Level, known to Thinklab as a subjective trait, to classify a continuous quantity according to an interpretation of the values. This style of model specification eliminates the need for defining concepts like “HighElevation” and “LowElevation.”. There are many predefined traits in the **im** ontology describing general attributes such as regularity of occurrence (regular/irregular), frequency of occurrence (ephemeral, rare, common or continuous) and origin (endogenous/exogenous). Learning to use traits appropriately is the best way to ensure data and model compatibility.

Identities managed by authorities

In most cases when an identity is used, there are many – and sometimes infinite – possibile concepts, and the attribution of identity is normally subject to great debate and change. For example, biological species have a many-to-many relationship with the taxonomic concept they describe: individuals of the same species may have been attributed to different ones before realizing that they were two growth stages of the same, and there is an enormous amount of species that grows every day. Chemical “species” work similarly: the periodic table of elements is relatively stable, but molecules are certainly not something we can classify in a single ontology, and even if so, we certainly would not want the humongous chemical ontology only to describe water and carbon dioxide.

Fortunately, we are not the only ones to need a stable reference framework for this kind of “open-ended” identities, and organizations such as the Global Biodiversity Information Facility have been established to provide exactly such framework. Such organizations provide a vocabulary and a process to assign a specific, stable identifier to a concept, so that it can be “tracked” unambiguously throughout the changes that it has undergone during its use in scientific practice. Similar organization promote unique ways to define molecular structures, agricultural terms, etc. Thinklab provides a way to define identities based on these identifiers, and the software we provide links some authorities in so that annotation becomes very simple and efficient. Authorities are identified by an uppercase string, such as GBIF, and the syntax for an authority-backed identity is as follows:

```
<abstract observable> identified as “<key>” by
<authority>
```

For example, to annotate a model that observes the number of individual of the fish species Argyrosomus hololepidotus, you can refer to the GBIF identifier for the species and write

```
count im.ecology:Individual identified as "5212442"
by GBIF
```

The part starting at **identified** counts as the definition of an identity trait, which is not specified directly as a concept, but by referring to an identifier managed by the GBIF authority. The authority name must be recognized and correspond to a plug-in installed in the language; use of authorities comes with the integratedmodelling.org certificate. Thinklab provides interactive search facilities and translation for some authorities: for example, the specification above will create a concept that displays (for example in data legends) as the common name of the species. Authorities will be developed and made available to suit the user communities; at the moment the three authorities available are

1. GBIF for taxonomic identifiers, as above;
2. AGROVOC (from FAO) for agricultural identifiers; and
3. IUPAC for chemical species identified by an InChl string.

Only the GBIF authority has search facilities associated for the time being. For all others, identifiers can be retrieved by using the institutions’ web sited. Requests for supporting other authorities can be sent atintegrated.modelling@gmail.com.

Inherent qualities and subjects

In ontologies, properties are used to classify the type of relationship that exists between concepts. The “specialization” property is frequently used (e.g., a **Car** is-a (type of) **TransportationVehicle**) but there are many others, both very general (e.g., **Individual part-of Population**) and specialized (**StreamReach has-slope Slope**). Although properties can be explicitly detailed in a concept specification, Thinklab can create and validate properties automatically, to reduce the complexity for the modeler and encourage shorter, more readable specifications. Yet, there are some important logical implications to consider before connecting concepts and making observations. One of which is determining the legal and illegal properties of a subject: a Watershed has Rainfall and Elevation but no Liver or Heart. Without having to explicitly create such constraints, which would be difficult, it is possible to specify the allowable subject observables to refer to, thereby avoiding improper usage in the model resolution process. This type of specification is referred to as an inherency specification, which in Thinklab is created using the keyword **within**. For example:

```
model ... as measure im.geography:Elevation within
im.geography:Region in m;
```

Assume there is a dataset specification (as we will see in Section 2) instead of an ellipsis, and consider only the observable of the model: a quality (elevation) with an inherent subject (a geographical region). This guarantees that any observation of **Elevation** will use data only if the context of the observation is a **Region**. Note that this works for any kind of region: for example, both a Watershed and a Country are specialized types of Regions, so the data will satisfy a request for observing elevation in both of these instances. Observing the elevation of a tree, however, would not be possible using the data produced by the model above.

Inherent subjects are used to restrict the semantics of the model to an appropriate set of applications.

Inherent subjects are used automatically during resolution: if there are two datasets for a concept, one is inherent to a Region and the other is not, the dataset inherent to the region will be given precedence during model resolution (seeModule 5) for full details on prioritization in resolution). In general, there should always be an inherent subject for all observables; but the choice should be made intelligently, as it is always possible to choose a subject so restricted that models become almost useless. For example, identifying a `LowerAtmosphere Region` as the inherent subject for Temperature data may make them less useful than simply using a Region, given that the "default" meaning of temperature data refers to the Earth's surface. Both specifications, and probably many others, may be conceptually correct, but the conceptual resolution of a model requires careful thinking: over-specification of semantics should be avoided as much as under-specification. If in doubt, remember that the same data can be annotated as many times as necessary, and there is no reason not to create both models if both observables have enough generality to be useful and sufficiently distinct meanings to be both useful.

Putting everything together

To summarize, an observable is composed of:

1. one and only one observable concept - a quality, thing, process or event;
2. if the observable is abstract, one and only one identity that grounds it to reality. For example, a species for an individual or group: `im.agriculture:Cattle im.core:Group`.
3. zero or more attributes and/or realms to complete the meaning of the observable if necessary;
4. zero or one inherent subject type, which specifies the most general kind of subject that this observable may refer to.

It is important that all the necessary concepts, and not one too many, are included in each model. Thinklab, as illustrated in the examples throughout this documentation, provides syntax to simplify the definition of an observable (in a nutshell: just string together trait concepts with their observable concept, and use the `within` keyword to introduce the inherent subject type if one is present). Consider the observable, in its three conceptual dimensions (observable/traits/inherency), as the "semantic fingerprint" of the model or data being described. Decomposing the observable keeps the ontology small - a parsimony principle which is crucial to the usability of a collaborative modeling infrastructure. The smaller the ontologies, the more useful and powerful they will be. Using separate concepts instead of specialization to capture the key meaning of observations avoids the explosion of the knowledge base and facilitates opportunities for more modelers to contribute to it.

Module 3. Connecting data to models: semantic annotation and observation semantics.

So far, we have seen that models produce observations of observables, which can be specified using a concept, identified with an identity if abstract, and optionally augmented with one or more attributes or realms and an inherent subject. All observations, except the "root" one made with the `observe` statement, happen in the context of a subject. A model therefore represents a strategy to observe a concept in a context. The result of the observation depends on the observable type (i.e., subject, process, quality, or event): observing a quality results in "data" being produced to represent a state, which will be distributed over the scale (space/time) set for the subject.

Writing models is the way to extend the power of Thinklab. The more inclusive the model library, the better the ability of the artificial intelligence engine to select an appropriate model for the specified context that best represents the observable. This definition of a model has the following consequences:

1. "Raw" numbers can never be used to represent the result of an observation in Thinklab: every number will always "be" something, i.e., have an observable associated with it.
2. Models 'annotate' values, data sources, equations, or external computations using the same syntax: in other words, data and models are merely two different ways of observing an observable (data are models).
3. In the case of computed models (e.g., equations), "inputs" are expressed as concepts. Thinklab resolves the concepts at run time based on the context.

Writing models consists of the following main steps:

1. Choosing the semantics for the observable.
2. Choosing and describing the "source" for the result of observing the observable: e.g., a value, data set, equation, external program.
3. Choosing the observation semantics, i.e., the type of observation made. In the case of things this is trivial, as these are direct observations, semantically equivalent to simply acknowledging the thing observed. For qualities, observation semantics requires more specification, for example of measurements, classifications, rankings, as will be seen later.
4. Creating metadata to help Thinklab track the provenance (i.e., origin) of the information during resolution, and choose the proper model when more than one model is available for the same concept. This will also be discussed in detail in Module 5.

Based on these principles, we see that an observation can be made in different ways, including extracting numbers from a data file, computing an equation, or calling an external program. This section only refers to resolvedobservations, where the states of the observation (numbers, categories, etc.) are known at the time the model is written: this applies equally to simple data values as to external data files, databases or data retrieval services. Module 4describes computed observations, which depend on computations.

Choosing a concept

Choosing a concept is a fundamental topic that is covered more extensively in the full documentation. For the purposes of this guide, it is important to remember that extending Thinklab's data and model integration capability depends entirely on the reuse of the shared knowledge base. As each Thinklab namespace is an ontology, concepts may be created at any point and by anyone, and nothing prevents a modeler from creating a new concept per each model without thinking of integration. But in a collaborative environment, new concepts should only be added when absolutely necessary and with community agreement on terminology and meaning. Many concepts can be created by combining existing observables with traits and inherent subjects as explained in Module 2. For the purposes of this discussion, the reader is reminded of three key points:

1. Understand and properly use the fundamental types of knowledge: subjects, qualities, processes, and events. Errors in attributing these fundamental types are certain to lead to trouble, both when running models and when interpreting outputs.

2. Learn to use traits, fundamental physical properties, and inherency. When in doubt, use a temporary concept that can easily be traced, and ask the larger modeling community for feedback.

3. Be mindful of common mistakes in attributing semantics to either data or models. A list of common ontology-related misunderstanding is in development to be integrated with this documentation.

Choosing the data or subject source

Models may have a pre-existing source of information for the semantics they provide. These are referred to as resolved models. Ultimately, all observations must end up as resolved models for each model input. Information sources can be provided for both data and subjects: examples include the value of a constant (e.g., the gravitational constant g or the boiling point of water), data from datasets (e.g., a precipitation or population density map), or subjects from datasets (e.g., villages, watersheds, or roads). Examples of each are provided below to illustrate how the `model` statement uses semantics to dress a "bare" reference to different kinds of data (all specified immediately following the word `model`).

Values

```
model 100 as measure im.chemistry:Water
im.physics:BoilingTemperature in Celsius;
model false as presence of im.theology:Satan;
model im:High as classify (probability of
im.climate:ClimateChange) by im:Level;
```

All these statements (which definitely belong to subjective scenarios!) show how the bare value of a quality can be set to a constant, which the resolver will take as the value in the context of validity of the previously specified model. Normally this form, which shows the most direct example of semantic annotation of data, is only used when testing or when creating scenarios, and most likely only for parameters that models should use under carefully controlled conditions: annotating constants as shown above will rarely be used in other ways. More typically, data annotations point Thinklab to data sets, which can be stored externally or directly stored with the other files in a project. This is done in the language by using functions that can define both data and subject sources, as shown below.

Data sources

Functions are identifiers followed by lists of named arguments within parentheses:

```
<function-name> ( <argument-name> =
"parameter_value", …)
```

The list of arguments may be empty, but if it is not, each argument will have a name and a value separated by an equal (`=`) sign. The function names and parameter names are not keywords of the language, so they may change and new functions may become available at any time. Thinklab provides a variety of functions, capable of bridging to several commonly used file formats and web-based data retrieval services; each function has its own argument names and rules for validation of arguments. To date, the most commonly used functions in Thinklab connect to spatial data. The following examples detail functions to access raster data from a Web Coverage Service (WCS) and from the filesystem on a user's computer, respectively:

```
model wcs(urn="im:global.geography:dem90m", no-data
= -32768.0) as measure im.geography:Elevation in m;

model raster(file="data/landcover.tif") as
      classify im.landcover:LandCoverType into
            im.landcover:Urban        if 200,
            im.landcover:Agricultural if 201,
            ....
            ;
```

Like the 'raster' function above, the 'vector' function can also be used with a 'file' argument to point to a file stored on a local disk within the user's project directory. Employing local files limits opportunities for collaboration and sharing, and it is very onerous to handle for the modeling engine. It is therefore recommended that they should only be used during model testing and development. Importantly, models that refer to local data files should not be shared in the integratedmodelling.org knowledge repositories, as all the data sources referred to in shared models must be accessible to everyone who shares the model itself.

Vector data often specify subjects, like roads or bridges, but sometimes they define distributed qualities in a more compact data storage form than raster data. For example, it is common to find vector representations of land cover type, although each polygon in the coverage is not a "subject" in a strict sense. Thinklab can use such data for qualities, as long as an attribute in them contains information in a recognizable form. In the following example, the attribute luc_id from the AFRICOVER vector dataset for Tanzania is accessed using the Web Feature Service (WFS) function:

```
model
wfs(urn="im:af.tz.landcover:tanzanialandcover",
attribute="luc_id") as
      classify im.landcover:LandCoverType into
            im:Agricultural if "AG",
            ....
            ;
```

When the resolver decides to use vector data in a grid spatial context, it will automatically rasterize the polygons, extract the value of the `luc_id` attribute, convert it into the concept indicated in the classification, and attribute it to each point of the grid. A numeric attribute would be required to provide a value for a numeric quality like slope or rainfall quantity. Note that the states of the observation will be `unknown` (a kewyword that expresses the notion of 'no data' in Thinklab) where no polygon covers the context.

Subject sources

In many cases, data sources can be seen as providing things rather than qualities. For spatial data, a common occurrence is vector files (such as shape files) where each record represents one distinct object, such as a road or a bridge. Not all vector files represent objects – some just use a vector representation as a convenience to lump together qualities that have the same values – but many do. In such cases, the models can annotate the sources as subject sourcesusing the keyword `each` and avoiding the observer statement after `as`:

```
model each wfs(urn =
"im:global.infrastructure:global_rail_merge")
named railroad-global
as im.infrastructure:Railway;
```

When located within a subject model (with the `each` keyword and only a subject concept after `as`), the source will be interpreted as a source of subjects. Observing the `im.infrastructure:Railway` concept in a context covered by the data above will generate as many railway subjects as there are in it, clipped to the context as necessary.

There is much more to be said about subject models, specifically about quality models that can be automatically inferred from them. We will briefly discuss some examples after the discussing observation semantics for quality models. Another topic that will be discussed farther along is how models can be written to specify what to do when the subject is built – which enables what is commonly called agent-based modeling in Thinklab. We will leave details on this advanced aspects for a further section.

Observation semantics for qualities

Observing qualities produces what we usually refer to as data, i.e., information that approximates the value of the state of the observable by referring it to a known set or scale. This "external" reference system is what we refer to when saying that qualities produce indirect observations. For example, elevation needs to refer to a unit of measure such as meters before a model can observe "how many units" of elevation are in a given location. Consider a 1 x 1 km region of land. Observing the elevation quality at the 100-m resolution will produce as many of these "data" as needed to cover the scale of the context - e.g., 100 numbers. In Thinklab, this set of 100 numbers counts as oneobservation of elevation in that context.

Thinklab provides a number of observer statements that help the user to specify the system of reference for a quality observation. This is equivalent to specifying the observation semantics for the observation. In Thinklab the observation semantics are, in general, only described through the observer. So, for example, a concept called `Measurement` will not be found in the Thinklab ontologies and no concepts of this kind should be added. For all practical purposes, the semantics of the observable and that of the observation are independent, and this is an important founding principle in semantic modeling. Some validation steps are taken to ensure that observables are appropriate for the observation and some exceptions exist to the above rule; still, observation types are best thought of as not directly implied by the observable concept

Data models are typically used by Thinklab when a computed model defines a dependency for an observable. This will be shown in detail later, but many models will state a dependency like:

```
model ....
      as ....
      using
            (Elevation as measure
im.geography:Elevation in m) named el,
            ....
            ... instructions to compute the result
based on 'el';
```

In this case, the code instructs the Thinklab resolver to look for the most appropriate model that satisfies it. If a data model is available for the observable, it will be chosen preferentially. Otherwise a computed model will be chosen if available, and its dependencies will be resolved in the same way. Observations made in different units can be converted when their relationships are clear: for example, a dependency on a measurement of elevation in meters may be matched to a data model for elevation in feet, and Thinklab will automatically translate the units.

Thinklab provides semantics for the following observation types: ranking, measurement, count, valuation, classification, proportion, percentage, probability, ratio and uncertainty. Each observer type has a correspondent statement that must be used after the keyword `as` (e.g., 'as rank', 'as measure', 'as count'), in any model that has a quality as its observable. Observers create values for the states of the concept they describe that may be different for different observers. In the sections that follow, each observer type is explained, including a description of the values produced, a description of how these data models will be used when matched to a dependency, and examples of use.

Ranking

Rankings produce numeric values that may use a scale (e.g. 0 to 1 or 1 to 10) or be unbounded, may be restricted to being integer numbers, and are meant to describe qualities for which a higher rank means a higher "level" for the observable concept. They do not have units (they often translate what is referred to as "arbitrary units" colloquially), and as such they should only be used when measurement or valuation observers are not an option. Rankings are typically used to express preferences or survey data, where the quality described uses an arbitrary numeric scale.

```
model wcs(...) as rank ...;

model wcs(...) as rank ...:PerceivedDanger 1 to 5;
```

This annotation will produce floating point numbers or integers if requested. If a range is given, values outside of that range that are produced by the data source will generate a runtime error, indicating a mismatch between the data source and its intended semantics. Some data source functions may offer the possibility of restricting the output range. This

annotation will match any dependencies for a ranking of a compatible observable. Note that: * If a `rank`statement in a dependency does not specify a scale, both ranking models with and without a scale will match it; * If a `rank` dependency defines a scale, only rankings with a scale will match it; if the observables match, the scale is seen as a "unit" of sorts, so the matched ranking may have a different scale, and the values which will be converted to the scale of the dependency before being used; * Dependencies that define a scale will not be matched to rankings of a compatible observable that do not define a scale.

Measurement

Measurements indicate physical properties, such as mass, energy, or entropy. Currently Thinklab will only generate a warning when a measurement is defined for a quality that is not a physical property, to give the modeler time to define concepts and still be able to test models. However, properly heeding the warning means that no user should ever "publish" a model with that warning to a public namespace.

More information about physical properties (and their fundamental distinction in being intensive or extensive, which is very important for their aggregation) is available here. For this module, it is important to understand that physical properties are measured and quantifying the measurement relies on the use of a reference system with standard units (e.g., mm, kg, ha). Thinklab requires the user to specify units according to a well-identified syntax and will validate units in both the client and the server.

```
measure ...
```

This annotation will produce floating point numbers in the specified unit of measure. This annotation will match any dependencies with a compatible observable. Units will be converted as necessary. If the annotation is correct, compatibility of observables should guarantee compatibility of units, and this compatibility is enforced to some extent in Thinklab, but not yet to a level that guarantees full coherency and safety. Consider the points made previously about aggregation and the use of distributed extents in defining the observables to be certain that observables and units are properly aligned.

Note that scientific practice and published metadata are often sloppy in defining units, both in terms of syntax and in using units for qualities that are not actually measurements (e.g. "10 people" or "20 EUR"). A semantic modeling system cannot afford that, so the appropriate observers must be used to handle qualities that could be called measurements (Note: the two examples above are a `count` and a `value` respectively - see below). Additionally, the raw values of some spatial data may require a conversion factor (e.g., multiplying by 0.01) to express the data in standard units. Such conversions can easily be conducted during data annotation; for example:

```
model wcs(urn =
"im:na.us.climate.annual:annualprecip")
      named precipitation-annual-2007-usa
      as measure im:Annual
im.hydrology:PrecipitationVolume in mm
      over time (year = 2007)
      on definition change to [precipitation-annual-
2007-usa * 0.01];
```

When in doubt, and particularly if modeled values appear to be consistently 'off', check the metadata.

Count

Counts are often considered measurements in common practice, but they are semantically unique, as they refer to a set of countable objects of a common type, and define the very particular quality that comes from counting them. The Thinklab `count` observer is special because it requires its argument to be a subject and produces a different concept. For example, when counting im.demography:HumanIndividual, the resulting observable (a quality) will be im.demography:HumanIndividualCount. Two additional observers, 'presence' and in some instances, 'classify', may also produce new semantics. At a minimum, only the subject need be provided to the count observer:

```
model 1 as count Universe;
```

If, however, a count that is distributed over space, time, or both, is desired, a unit to define the extent of the distribution is required. For example:

```
model wcs(urn="aries:global-populationdensity-2006")
as
      count im.demography:HumanIndividual per km^2
      over time (year = 2006);
```

Using the `per` syntax dictates the unit that would otherwise represent the denominator if the annotation (incorrectly) specified the count as a "measurement … in people/km^2".

This annotation will produce floating point numbers, although this is a bit of a semantic blasphemy, as countable subjects should not normally maintain identity when split, so only integer values are semantically correct. The ability to automatically account for densities or rates when the context is spatial or temporal requires enabling fractional counts. Note that the quality resulting from this model will be represented by a different concept, created (if necessary) by appending "Count" to the subject concept. This annotation will match any other count of the same observable over compatible extents. Units will be converted appropriately and automatic aggregation will take place.

Value

It is common to see metadata referring to value observations as "measurements" of value - either monetary or in "arbitrary units." A semantic system needs to do better than that: value is semantically very distinct from the generic measurement. It entails both subjective and objective comparison between different quantities and the value system that underlies valuation is much more fluid and culturally dependent than a physical measurement.

Value is commonly assessed monetarily, but that is by no means the only way to observe value. Thinklab allows the specification of monetary or conceptual currencies.

Currently, the use of the value observer should be considered experimental and support for value conversion in Thinklab (i.e., in converting between currencies or within a single currency to account for inflation) is limited. The syntax allows currencies to be specified like so:

```
model ... as value of ...:PropertyParcel in
USD@2004;
```

Because values can be non-monetary, the specification of the currency does not need to be monetary. If a currency is given, it must be qualified by the year (or month/year) after the `@` character, as the definition of any currency is meaningless without an historical context (which is independent of the possible temporal context). If the currency is not monetary, a concept expressing the type of value should be specified (e.g. "Affection"). In all cases, some currency is necessary, and the semantics defined by this statement will reflect both the observable and the way it is valued.

Note that while Thinklab is expected to enable automatic currency conversion, this feature is in development and is not available yet. For all practical purposes, `value` will behave the same as a `measurement` without a scale even when a temporally-specific currency is specified.

This annotation will produce floating point numbers. This annotation will match any value that has the same currency and year, or the same currency concept. In the future, translation to other currencies and years will be provided.

Classification

Classifications are often used in modeling to define categorizable attributes of a common type (e.g., land cover type) or to "discretize" continuous values into discrete levels that can be more easily understood or fed into computations that require categorical data. In a semantic system, we need to unravel the meaning of categories, and using string values won't do us any good - but in the semantic world, the idea of "categories" corresponds very closely to ontologies and concepts. Indeed, the `classify` observer produces concepts, and it is carefully designed to be able to express all the semantic nuances described for observables: types, traits, and discretized levels. As a result, the `classify`observer has several options, although all should read naturally.

In a classification, the state of the indirect observation is a concept. This corresponds to what is commonly called a categorical observation. Because in Thinklab nothing can be devoid of semantics, it is not possible to produce categories that are simply strings (e.g. "high" and low"). Any categorization must have clear semantics, so any time models call for categories, a classification must define the categories in the terms of an explicit concept hierarchy.

Classify has three different forms, which are used in different circumstances. Direct and indirect classifications simply define the list of concepts that make up the classification, and are used when the concepts are produced directly in the model, for example through an algorithm. Classification using observables or traits is used when the concepts are produced by reclassifying some other observation; classification rules are defined to produce each concept. The latter form can use a trait as the basis for classification.

As a special case, the `classify` keyword can include only the observable concept when stating a dependency. This special case will be explained in Module 4.

Direct classification

In a direct classification, the concepts that form the concept space are listed directly after the observable concept and the keyword `into`:

```
model ManureType as
      classify im.agriculture:Manure into PigManure,
CattleManure, PoultryManure
   observing
            (PigManureProportion as proportion of
im.agriculture:Pig  in im.agriculture:Manure
im.core:Mass) named pig-manure,
            (CattleManureProportion as proportion of
im.agriculture:Cattle in im.agriculture:Manure
im.core:Mass) named cattle-manure,
            (PoultryManureProportion as proportion
of im.agriculture:Poultry in im.agriculture:Manure
im.core:Mass) named poultry-manure
      using rand.select(
            distribution = (pig-manure cattle-manure
poultry-manure),
            values = (PigManure CattleManure
PoultryManure)
      );
```

This model creates four concepts: ManureType, PigManure, CattleManure, PoultryManure in the namespace where the model is declared. The last three concepts are children of the first. The concepts are produced directly by the`rand.select` accessor (which selects one of the child concepts based on the probabilities specified by each of the observed dependencies) there is no need to specify any other criterion for classification. The specifics of the rand.select accessor are treated elsewhere.

Indirect classification

The indirect classification assumes that all the concepts in the concept space have already been defined and tagged with metadata. The metadata field is used to link the appropriate concept to the value extracted from a data source or observed by a mediated observation. The link is established with the `according to` keyword sequence followed by the metadata field that contains the linking value. Concepts with established numeric encoding, like those used in numerically classified land-cover or soil order data, offer significant time savings and limit the ability of the user to introduce error in the coding scheme:

```
model data.wcs(id = "europe:corine2000", no-data =
255) named corine-2000
      as classify im.landcover:LandCoverType
according to im:numeric-encoding;
```

This will only work if the children of the observable concept (in this case `im.landcover:LandCoverType`) are tagged in the respective ontology with an `im:numeric-encoding` field that specifies the number that will be extracted from the `data.wcs(...)` data source. An example LandCoverType ontology snippet is below:

```
...
class LandCoverType
      has children
            ...
            (DiscontinuousUrbanFabric with metadata
{ im:numeric-encoding 112 })),
            (class IndustrialCommercialTransport
                  has children
                        (class
IndustrialCommercialUnits with metadata
{ im:numeric-encoding 121 }),
                        (class RoadRailNetwork with
metadata { im:numeric-encoding 122 }),
                        (class PortArea with
metadata { im:numeric-encoding 123 }),
            ...
```

This approach transfers the burden of annotating the encoded value for a datasource from the model to the concept. This is worth doing if the concepts are used for more than one data source; otherwise the effort is the same and it's just a matter of stylistic preference whether to use this form or the mediating classification discussed next.

Classifying values into observables or traits

The most complete classification specifies classifiers that attribute the result concept according to results of the "incoming" information (e.g., the values that come from a dataset). A common use of classifiers is to annotate a data source:

```
model wfs(urn =
"im:af.tz.landcover:tanzanialandcover",
            attribute = "lc")
named tanzania-lulc
as classify im.landcover:LandCoverType into
      im.landcover:AgriculturalArea        if "AG",
      im.landcover:ForestSeminaturalArea   if "NVT",
      im.landcover:VegetatedStillWaterBody if "NVW",
      im.landcover:UrbanFabric             if "UR",
      im.landcover:WaterBody               if "WAT";
```

The classifier, the part that follows the `if` keyword in each row, can accommodate many kinds of expressions, discussed in detail in Module 4 and demonstrated further in the Cookbook. Numbers, strings or concepts can be matched using 'is', numeric intervals using a syntax like '1 to 10', or partial intervals using operators like '< 10' or '>= 4.3'.

Data often represent a classification according to a specific aspect of the

main observable, one that is best described with an attribute. In such cases, classifications should be annotated using the main observable and instead of defining its subtypes in the classifiers, use the 'by' keyword and mention the specific trait being observed. Then, the classifiers will be defined by the appropriate attributes. A typical case is when data describe a discretization in subjective levels:

```
model wcs(...) as classify im.policy:Poverty of
im.demography:HumanPopulation by im:Level into
            im:High     if 4,
            im:Moderate if 3,
            im:Low      if 2,
            im:Minimal  if 1;
```

Note that if data are available and the modeler wants a discretization of known numbers, the proper annotation is not **classify** but the actual numeric observer; models without data sources can be used later to discretize the value. For example, if a discretization of im.geography:Elevation is needed, do not annotate a data source as:

```
model wcs(...elevation data...) as measure
im.geography:Elevation in m
      discretized by im:Level into
            im:High if > 2000,
            ....                    (An INCORRECT
example)
```

This would make available a model of elevation data that ranks it into subjective values that only make sense for a specific application, losing the numeric information in the data. This model would match any dependency for elevation data, and produce discretized midpoint numbers when the requesting side wants numbers, with a great loss in precision. A proper way would be to just provide the undiscretized measurement in a public repository, and create a local model that translates it in the same namespace where it will be used, marked **private** to ensure that nothing else will use it:

```
namespace my.namespace using im.geography;

(RIGHT, BUT STILL NOT PERFECT)

private model Elevation as
      classify im.geography:Elevation by im:Level
            im:High if > 2000,
            ....;
```

We will see in Module 4 that this kind of requirement can be stated directly as a dependency, eliminating the need for a private model (and the risk of forgetting the **private**, making it available for others with potential problems) and keeping all the subjectivity nicely encapsulated within the context where it is needed.

Values in the data source that fall outside the space defined by the classifiers in a **classify** statement will appear as **unknown,** in the final observation, the Thinklab definition of "no data". This is sometimes a handy way of selecting only a few categories or values from a data source. Operations, such as sums or subtractions, which have an unknown operand will result in an unknown result.

Proportion and percentage

Proportions and percentages are two almost identical ways of referring to a quantity that represents a portion of an implicit "whole" amount. The semantics of the quantity and the total amount usually only differs by a trait, with the amount at the numerator being more specific than the one at the denominator - for example, the proportion of "vulnerable" land over the total land. Thinklab allows both to be specified:

```
model ... as percentage of im.agriculture:Pig  in
im.agriculture:Manure im.core:Mass;
```

In the case above, the quality concept is stated in the second part **im.agriculture:Manure im.core:Mass**: an abstract quality (mass) qualified with a Manure identity. The "specific" case that is compared with the less specific manure mass is identified by another identity after **in**. The whole statement read as "annotate these data as the percentage of manure mass that can be attributed to the Pig identity".

You can use the keywords **percentage** or **proportion** to annotate data that contain numbers in the interval [0-100] or [0-1], respectively. Thinklab will match proportions and percentages to each other, converting the numbers as necessary.

Some concepts may be inherently defined as proportions or percentages: for example, ecologists are accustomed to work with "canopy cover", which represents the proportion of land covered by the tree canopy in a forest when seen from above. In such cases, when you certainly don't want to think about the "generic ground" and the "canopy" that covers it, you can simply use the concept without specifying **of** and **in**:

```
model … as proportion im.ecology:CanopyCover;
```

Note that in such cases, the concept will be validated as actually expressing a proportion, and an error will be flagged if that's not the case. In other words, Thinklab will let you use a simpler semantics only if the semantic groundwork has already been done correctly in the ontologies you are using.

Ratio

Ratios are interesting because they don't have one observable but two. Indeed, our definition of an observable is something that is "grounded" in reality; while ratios only exist as comparative observations of two observables. There are no true ratios in nature. For this reason, we provide only the full form of the observer where both compared qualities are annotated independently:

```
model … as ratio of im.ecology:Soil
im.chemistry:Carbon im.core:Mass to im.ecology:Soil
im.chemistry:Nitrogen im.core:Mass;
```

Because there are no concepts that can be seen as natural ratios, we do not provide a simpler form as we did with proportions and percentages. This model will match any ratios of compatible observables and produce floating point numbers. An appropriate concept will be created by the annotation: in the case above, the concept will look a bit complicated – something like **im.ecology:SoilCarbonMassToSoilNitrogenMassRatio**. Fortunately you can state your dependencies using the ratio observer, and never have to write that.

Note that knowing the ratio of two quantities allows an intelligent software to infer one quantity when the other is known. So for example the above model should be enough to satisfy an observation of carbon mass in the soil when a model producing the nitrogen mass in the soil exists. This capability is in development in Thinklab.

Presence

Presence is the quality corresponding to a subject, process or event "being there" in the context. As such, it can only take the values **true** or **false**. Presence is the observational equivalent of a "boolean" value in data-oriented languages.

Thinklab provides a simple statement to annotate presence data:

```
model … as presence of im.infrastructure:Building;
```

Note that the concept after **of** must be a thing and can not be a quality, as there is no such things as "presence of a temperature" for example. If you need to know, for example, whether temperature is above 25 Celsius, the statement you are looking for is not a **true** or **false** statement, but a

classification of a restricted conceptual hierarchy that classifies temperature into “above 25” or “below 25”.

Like in proportions and percentages, it is possible to encounter concepts that are semantically defined as types of presence, so the form without the **of** is admitted with validation. Note, however, that this is rare and the form above is the most common one.

This model will match any dependency requiring presence of a compatible observable. The values resulting from it will be **true** or **false**, and a quality concept such as **im.infrastructure:BuildingPresence** will be created if not already existing. Note that if you have annotations of the subjects themselves (e.g. a vector file of buildings) it is usually not necessary to provide a presence annotation, as that is automatically inferred by Thinklab: see the section on de-reification below for details.

Probability and uncertainty

The last two observers refer to the observation of the likelihood of something happening or being observed correctly. They are provided to simplify annotation of risks and to make it easy to track model and data uncertainty.

The probability observer has both the explicit form

```
model … as probability of im.physics:Fire within
im.ecology:Forest;
```

and a simpler form for concepts that are naturally probabilities, such as risks. In the explicit form, the observable after **of** must be something that happens: an event or a process. Again, you may be tempted to conceptualize probabilities of qualities or subjects as semantic shortcuts, but Thinklab will not allow that. The observer will produce numbers in the range [0-1] and will flag as errors any data output that falls outside this range. The direct form will produce a concept such as **im.physics:FireProbability**.

The uncertainty observer refers generally to the spread of a value around its maximum likelihood estimator, but does not mandate any specific type of uncertainty: simply the meaning for it, producing a numeric expression that means “less certainty” when it is higher. It is used often to annotate expected outputs from external models that produce estimates of uncertainty along with values, such as Bayesian networks. It is more rarely used to annotate data, as few direct estimates of uncertainty are available. Like probability, it has a direct form with **of** and an indirect one for observables that are defined as uncertainties. Unlike probability, it has no constraints on the type of observables that can be used: you can estimate the uncertainty of anything, including another uncertainty. Also unlike probability, it can produce floating point numbers in any positive range. The observer will create a concept such as**im.ecology:CanopyCoverUncertainty** if that does not exist already.

Discretization

All numeric observers (**rank**, **measure**, **value**, **ratio**, **proportion**, **percentage**, **count**, **probability** and **uncertainty**) can be discretized into discrete levels. The most common way to do it is by using a trait, often usingim:Level or a trait that derives from it:

```
model wcs(...) as measure im:Length of
im.ecology:Leaf in cm
      discretized by im:Level as
            im:Low    if < 10,
            im:Medium if 10 to 30,
            im:High   if > 30;
```

In this case, the output of the model is a concept as in a classification, but the quantitative meaning is not lost: when used in a numeric context, the values will be automatically converted in the midpoint of the intervals as long as the boundaries are finite (they are not in the example above). When specifying a discretization, the classifiers must be continuous (the endpoints must touch): the convention for intervals specified as above is that the interval is closed at the beginning and open at the end.

Note: spatial densities and temporal rates refer to observations, not observables

It is important to distinguish between the primary identity of what is to be observed and those secondary aspects of the observation that depend on being distributed over space or time when defining the semantics of an observable. It is common to encounter concepts like “population density,” for example, whose definition as a “density” depends on the observation being made over a spatial context (e.g., “per hectare” or “per square kilometer”). In Thinklab, space and time are part of the context; for this reason they don’t enter the semantics of an observable, but are automatically handled when they are observed over space, time, or both. Confusion in semantic modeling is likely when the common attribution of concepts used in data or models implies that they are distributed over a specific type of context (spatial or temporal), even though they may be used in others. So for example, a concept named PopulationDensity (which implies that applied to the population observable is distributed in space) should be avoided, in favor of the generic semantics of population without such implications. In Thinklab, the appropriate observable for this case is acount of people, expressed for example as

```
model .... as count im.demography:HumanIndividual
per km^2;
```

Only subjects can be counted, so this model, which contains all the semantics of what population density is (a count of individuals over space),requires a subject concept (im.demography:HumanIndividual). The model implies a density, but only when the data source is spatially explicit. Thinklab will, if necessary, aggregate densities automatically into total numbers of individuals when the model is matched to a context that is spatial but not distributed in space (e.g. a city defined with only a single polygon without grid cells). The same applies to rates, which are the equivalent of densities with respect to time. Observables and observations are independent, and their semantics should not be “contaminated” by concepts that have to do with the characteristics of the context of observation. Avoiding references to spatial densities and temporal rates will maintain semantic consistency and help to minimize confusion.

De-reification of subject models

Having discussed quality models in detail, we can go back for a moment to subject models. Observations of subjects are direct, so it is not necessary to state anything more than the subject type. Direct observations simply create subjects. So an object source such as a shapefile can be annotated to create subjects as seen above. In addition, the existence of a subject also implies certain qualities: an obvious one is the **presence** of the subject, which takes the values **true** or **false** according to a subject being in a point of the context or not. Thinklab will automatically generate quality observers for presence whenever a subject annotation is encountered, so that observation of **presence of im.infrastructure:Railway** will be made automatically by rasterizing the line contexts coming from the subject annotation above, and returning **true** for each point where a railway is present.

Further, semantics can also be used to specify the qualities referring to each subject. Thinklab expects that subjects produced by a subject source

may come with attributes, Adding observers (see below) to the subject will determine their qualities. Consider this example:

```
model each wfs(urn =
"im:global.infrastructure:grand_reservoirs_v1_1")
named reservoirs-global
as im.hydrology:Reservoir
interpret
      GRAND_ID as im:name,
      AREA_SKM as measure im.core:Area in km^2;
```

In the example above, the `interpret` keyword introduces a list of attributes and the ways they should be interpreted when creating subjects. As each reservoir subject comes with an attribute (`AREA_SKM`) that contains a measurement of its area in its original source, we specify the second attribute with a quality observer for it. This has two effects in resolving dependencies:

1. any dependency for a Reservoir subject that is satisfied by the subject source will create reservoirs that contain the quality of Area;
2. a quality dependency such as `measure im.core:Area within im.infrastructure:Reservoir …` will be satisfied by a model inferred from the above specification. The model will be matched to the context of observation: for example, when asked to measure reservoir area over a spatial grid context covered by the subjects above, Thinklab will automatically rasterize the `AREA_SKM` attribute and if necessary, convert the values in the units requested.

This ability of creating qualities from things is called de-reification (removing the “thing-ness” in things) and is very useful to make the most out of data. You could also annotate each quality explicitly:

```
model wfs(urn =
“im:global.infrastructure:grand_reservoirs_v1_1”,
attr=”AREA_SKM”)
      named reservoirs-area-global
      as measure im.core:Area within
im.hydrology:Reservoir in km^2;
```

But this is made unnecessary by the subject annotation, which provides this and an automatic `presence of im.hydrology:Reservoir` along with the subjects themselves – three birds with a stone.

You may have noticed the `GRAND_ID` attribute, annotated as just providing a single metadata property, im:name. That will not produce any observation, but tells Thinklab that the subjects should have metadata reflecting the content of that attribute. You cannot observe a name – no physical reality in that – but each reservoir produced can be given the name specified in the attribute, a useful property for visualization. That is the equivalent of the name you specify in an `observe` statement – subjects have individuality, so they can have their own metadata.

Module 4. Computing deterministic and probabilistic observations.

What is most commonly referred to as a model (outside of Thinklab/ARIES) is actually an algorithm that computes quantities that describe “virtual observations,” which reflect a hypothesis about the observable that is expressed in equations or other processes. Such modeling relates the state of the observable to that of other observables. For example, we can interpret tree biomass as dependent on precipitation, soil type, and incoming solar radiation present or past. With this in mind, we can define the two main differences between “data” and “models” in Thinklab. Models, but not data, can:

1. Define an equation, algorithm, or external process to compute the state of the observable;
2. Have dependencies, i.e., the statement of other observables that need to be observed before the state of the model’s observable can be computed using some type of algorithm.

These two points introduce additional specification requirements, rather than structural changes, when compared to data models, i.e., data, in Thinklab. We have defined data as resolved models before, because data don’t require any other observable to be computed before them. Therefore it is guaranteed that data will produce observations when used in a compatible context. Because the definition of a computed model is just like that of data with some added specifications, we use the `model` statement to annotate both data and models. Data are effectively a pre-computed model of the observable, which does not require any further observations but is, like any model, dependent on scale, assumptions, etc. Data models may also include equations, which are useful when, for example, when we wish to modify raw data before it is used as the state for an observation.

Because Thinklab is a semantic system, each dependency is defined semantically, i.e., by providing the details about the identity of what needs to be observed, not by giving the system equations or other information concerning how to compute it. For this reason, models in Thinklab are usually small; the final, integrated algorithm that computes a top-level observable concept (e.g., “climate”) is defined by resolving each dependency to models, a process that may bring in new dependencies as models are chosen. Observation of a complex observable is successful when all the “end points” of the model chain are resolved models. At that point, Thinklab can create a data flow from the model chain, and run that to compute the states of every observable involved, creating the final subject that contains observation for qualities and other subjects as its semantic constraints require.

Model syntax: observable, dependencies and computations.

A simple computed model may sum the state of its dependent qualities to obtain the state of its own quality observable:

```
namespace my.namespace using im, im.agriculture;


...


model SummerCropYield as
      measure im:Summer im.core:Yield of
im.agriculture:Crop in t/km^2
      observing
            (JunCropYield as measure im:June
im.core:Yield of im.agriculture:Crop in t/km^2)
named jun-yield,
            (JulCropYield as measure im:July
im.core:Yield of im.agriculture:Crop in t/km^2)
named jul-yield,
            (AugCropYield as measure im:August
im.core:Yield of im.agriculture:Crop in t/km^2)
named aug-yield
      on definition set to [jun-yield + jul-yield +
aug-yield];
```

There are several new things to note in this example:

1. Instead of a data source, we specify a name after the `model` keyword, using the capitalization conventions used for concepts. Indeed, we are creating a concept. Essentially this model is an ontological statement: we define the concept `my.namespace:SummerCropYield` while giving an interpretation of how it should be computed (i.e., as the sum of crop yields in June, July, and August). As noted in the introduction, each Thinklab namespace is an ontology: the `im.namespace` ontology that is created by Thinklab reading this statement contains the new concept.

2. A new keyword, `observing`, is followed by a comma-separated list of dependencies. Each dependency is a mini-model itself, but without computational details. Within each parenthesis, there is a concept ID for the observable and the full semantics of what is observed. The keyword `named` is used to give the dependency a name, which will only be used within this model and won't be recognized by other models, even those in the same namespace. This name is used to refer to the value of the observation in equations and conditions. The `im.namespace` ontology will also contain the three concepts corresponding to the first identifier, and a dependency relationship that links them to the main observable (SummerCropYield).

3. The last line, whose syntax we will describe later, shows the simplest way to compute a quality in Thinklab: an expression, defined within square brackets, that returns the value of the observation. In this simple expression, we use the names we defined above to refer to each operand (jun-yield, jul-yield, and aug-yield).

The context of applicability for a model

At this point, it may be unclear how this model handles time and space, because there are no time or space specifications in it. In Thinklab, when temporal and spatial constraints are not specified, it means that the model applies to any time and any location. In data models, the data source may implicitly contain a temporal and/or spatial context, which automatically becomes the context of validity for the model. For example, if an annotated dataset contains a raster map of elevation that covers Spain, Thinklab will not use it to resolve a context located in Greece. But the SummerCropYield model doesn't have a data source, so the model, as stated, covers every situation: not only every part of the world, but also any definition of SummerCropYield that has no associated space or time. Methods to instruct Thinklab on where and when this model should apply are covered in Module 5.

There is, however, one other important consideration to correct the SummerCropYield model presented above. The units used throughout the module are t/km^2 - implying a dependence on space and no dependency on time. So if this model was used in a non-spatial context (for example created with an `observe` statement without an `over space ...` clause), it would produce the wrong results, returning a spatial density of yield (in t/ha) when a total yield (in t) is wanted. Indeed, the proper way to specify this model is

```
model SummerCropYield as
      measure im:Summer im.core:Yield of
im.agriculture:Crop in t/km^2
      observing
            (JunCropYield as measure im:June
im.core:Yield of im.agriculture:Crop in t/km^2)
named jun-yield,
            (JulCropYield as measure im:July
im.core:Yield of im.agriculture:Crop in t/km^2)
named jul-yield,
            (AugCropYield as measure im:August
im.core:Yield of im.agriculture:Crop in t/km^2)
named aug-yield
      over space
      on definition set to [jun-yield + jul-yield +
aug-yield];
```

Adding the `over space` statement, without further specification, ensures that whatever the usage of this model, it will only apply to a context where any kind of space is defined. In anticipation of Module 5, it should be fairly intuitive that following the 'over space' statement with a polygon, as in an `observe` statement, will also limit the use of the model to contexts that cover that particular space.

Importantly, because a quality has a value over the whole context, this computation will be repeated as many times as necessary to cover a distributed context. So if, for example, this model is chosen to compute a dependency on SummerCropYield in another model that is run over a 10x10 km spatial grid using a 1 km^2 resolution, the equation will be computed 100 times (once for each of the 100 grid cells), each time with the value of the dependencies computed by another model or data in the same cell.

The identification of an observable as a density (e.g., SummerCropYieldDensity) should be avoided for the reasons discussed in Module 2 and further explained in the discussion on scale in Module 5. SummerCropYield should be called a density if the observable was

semantically a density, independent of the context; for example, the density of water. The fact of being distributed over a context that is aware of space makes the observable a density only when observed in that context, not by nature. Aggregation, which in Thinklab is managed automatically, would remove the density "character" from the observation. The same considerations apply to time: if distributed over time, the units must have a temporal unit in the denominator, but it would be inappropriate to name the concept a SummerCropYieldRate.

The observable in computed models

The most obvious difference between a "resolved" model that annotates data and one that is "unresolved" model is that the latter includes an observable concept definition after the `model` keyword. Indeed, a computed model creates new knowledge (or "reinterprets" it), and the concept statement defines its semantics. The concept may be defined before the model, or more than one model may be provided for the same concept; the first time an unknown name is encountered, a concept will be created with the semantics that depends on its use. Successive uses of the same name will need to be consistent with that semantics.

In some situations(detailed below), this concept can be written without a colon-separated namespace identifier, which we use to qualify an external ontology in the Thinklab naming conventions. This defines a concept that is "local" to the namespace within which the model is defined. So the model in the example above will in fact create the my.namespace:SummerCropYield concept. If the model has an observer, this concept will be a specialized version of the observable defined for it. So SummerCropYield will be in a "is-a" relationship with the SummerCropYield concept created by the `measure` statement inside the model, and will inherit all the traits from it.

Compare the model above to a "resolved" data model described in the Module 3: only the observer semantics is stated for a resolved data model, because the data source takes the place of the top-level observable. Indeed, we areannotating data so that specification is all we need. When we compute data, we are defining new knowledge, hence the need for a concept after `model.`

When the primary observable is local to the namespace, we expect that the concept will only be useful inside of it, for example to use as a dependency in other models in the same namespace. This is done commonly when a namespace contains models for many different observables, all of which are used in a final model that brings the whole conceptualization together.

In other situations, the models can use a primary observable concept that has been defined outside the namespace. This choice is normally made when the namespace is meant to provide an "alternative" way of observing a concept that is localized to a particular scale – for example to a given geographical region. In such cases, observations of the observable will only be made with these models where the constraints are met – which may even be only a subsection of a full context of observation. This is discussed more in detail in the next sections; however, just note that models with a "known" (external) observable are typically useful in a shared context, while models with "local" observables are usually written to provide clarity and internal organization in namespaces that want to keep most of their models private. In the examples below, we only create knowledge in the local namespace, which can be used in outside models only when qualified with the full name for the concepts (e.g. my.namespace:Elevation).

Mediation

Mediation is mainly used with classifications. It is a way to chain different observers together for use in models. Mediation introduces an implicit dependency for "another view" of the same observable. A typical example is:

```
namespace my.namespace;

...

model ElevationLevel as
      classify (measure im.geography:Elevation in m)
by im:Level into
            im:High if > 1000,
            im:Low  if < 1000;
```

This kind of specification nests a dependency within an observer, without assigning it a concept and a name. The example defines a classification that depends on observing a measurement of elevation. The part within parentheses is indeed a dependency, but because the semantics of the "inner" observation relates directly to that of the observable, and we don't need to preserve the value of the measurement for any computation, we can more simply set it in place of the concept to be observed by the classification. This kind of statement is a shorter, more intuitive and a more "fluent" idiom than the equivalent model expressed with dependencies. While the model below will produce the same results as the one above, it is longer, more complex, and less readable, so the above model would be highly preferable:

```
model ElevationLevel as
      classify ElevationLevel by im:Trait into
im:High, im:Low
      observing
            (Elevation as measure
im.geography:Elevation in m) named elevation
      on definition set to [
            elevation < 1000 ? im:Low : im:High
      ];
```

For this reason, mediation should always be used when it's appropriate, in place of a less readable model with one dependency.

Expression language

The next few sections provide some guidance on how to write the expressions between the square brackets and where to learn more about them. The language used for these expressions is parsed by a Groovy interpreter, after being pre-processed by Thinklab so that concepts and identifiers known to Thinklab can be used without error.

Groovy is a superset of the Java language, with less strict syntax rules and optimized for use in scripts and expressions, but containing all the power of Java plus numerous enhancements. More details about the language are outside the scope of this guide, but it is important to know that the language is much more powerful than the simple expressions we use as examples can show. Exploiting this power obviously requires programming skills. For the interested, there are many resources about the Groovy language, both on the Web and in print.

The main things to know about the way Thinklab and Groovy interact are:

1. The identifiers used after the `named` keyword in dependencies can be used in Groovy as variables that contain the value being computed for that dependency (in each state determined by the scale, e.g., in each grid cell, with one expression evaluation per state). The naming conventions used in Thinklab (lowercase and dash-separated) are not compatible with Groovy, which does not use dashes within identifiers. The names that are declared are automatically substituted in the expression before Groovy sees them, but if the wrong identifier is provided (i.e., due to a typo) that contains a dash, Thinklab will not substitute it, and Groovy will interpret that as a subtraction of two variables, yielding a potentially confusing error that will hint at only one part of the variable name being undefined (e.g. “population” when the expression contains “population-density” but that has not been defined as the name of a dependency).

2. The “no data” value, indicated in Thinklab with the keyword `unknown`, is the equivalent of “null” in Groovy and Java, and will be translated to that before the expressions are passed to Groovy for calculations. If an expression has a chance of encountering an `unknown` value (e.g., from no-data points in a data source) it should be prepared to handle it. Groovy allows null values to be added and subtracted, making the result null without error; but multiplications, for example, will break the computation and produce an error. We are working on solutions to this problem, but for now statements may need to be “null-proofed” like this:

   [(a == unknown || b == unknown) ? unknown : a*b]

The ternary operator above (C ? Y : Z) will return Y if the condition C is true, or Z if it is false. In English, it is the equivalent of saying “is a unknown or b unknown? then return unknown; otherwise return the product of a and b”. Either `unknown` or `null` can be used interchangeably. Note the two equal signs - Groovy, like most languages, uses a single equal sign for assignments to variables, and the equality operator is `==`. If you need more complex expressions, please refer to Groovy documentation for details on syntax, which is compatible with the better known Java, and for functions you may call (e.g. mathematical functions).

3. The Groovy type of the values assigned to dependency names will be 1) a floating point number if it comes from a numeric observer (rank, measure, value, count, percentage, proportion, ratio, uncertainty); 2) a boolean (true/false) value if it comes from a `presence` observer; or 3) a concept if it comes from a `classify` observer. For those who can negotiate a Java/Groovy interface, concepts conform to the IConcept java interface described in the Thinklab API documentation. Proficiency in Java/Groovy allows access to many functions related to concepts, but for the uninitiated, it is usually enough to know that the operator “is” can be used for a concept, followed by a literal concept identifier, like so:

   [(land-cover-type == unknown || land-cover-type is im.landcover:Urban) ? 10 : 20]

The Thinklab cookbook included in this documentation contains examples of expressions of common usage.

Using expressions in data models

Expressions can be used to modify or create the value of resolved models as well. It is quite normal to link to a data source that requires some processing before it can fully express the desired observation semantics. In such cases, a model this can be used:

```
  model wcs(...)
       named ghg-emissions-usa
       as measure im.policy:GreenhouseGasEmissions in
t/ha*year
       on definition change to [ghg-emissions-usa *
0.0001];
```

The above model will work as a data annotation, but will transform the value into a tenth of a thousand of what the WCS data source contains before producing the observation. Similar expressions can be used for more complex transformations or for filtering of values. Models can have both data sources and dependencies. Each model dependency will itself need to be resolved, but the final value can be assigned using expressions that take into account both the value of the data source and that of the dependencies.

Limitations

Processing of expressions in Thinklab is still fairly primitive, so there are several limitations that will be removed in future versions. The most important are:

1. Square brackets ([,]) can only be used within an expression if the closing bracket is quoted using a backslash character, as in `\]`. If this is not done, the closing bracket will be interpreted as the end of the expression. This makes the use of the standard array notation with Groovy a bit awkward, although this is only of concern for modelers with coding skills. This limitation will be removed in the future.

2. The pre-processing of Thinklab concepts and symbols within expressions uses a fairly unsophisticated pattern matching algorithm, which may fail when identifiers contain other identifiers (for example, if dependencies have very simple names like “a” or “b”). When in doubt, use longer, more descriptive names and provide space around all identifiers. We will soon switch to a parser that has no such limitations.

3. The content of the expressions is not analyzed by Thinklab in the same way that rest of the model syntax is. The expression code is only parsed the first time when the model is run, so it is possible to write completely incorrect expressions that will not show errors in the editor yet produce errors at runtime. When models containing wrong expressions are run, errors will be reported that should be easy to relate to the offending expression. In the future, the expression parser will be integrated more deeply in the language so that syntax errors in expressions can be seen during editing.

Of course, no language analysis can identify logical errors in the expressions. So always test models with particular care and under different scenarios of use when they contain expressions.

Dependencies in detail

As we have seen above, a list of dependencies is introduced by the keyword `observing.` In order to compute the observable, the dependencies in the list must first be observed. This applies to both qualities and subjects, as it may be desirable to observe subjects (e.g., households or bridges) in order to observe a larger-scale subject, for

example a region, that contains them. On the other hand, remember that any dependency is also a semantic statement of a relationship between the observable of the model and that of the dependency. Based on our definition of a quality, it is pointless to depend on subjects when the observable is a quality. For this reason, Thinklab does not allow qualities to be part of a list of model dependencies, although qualities held by subjects may be referenced through the de-reification mechanism discussed above for subject models.

One general, but advanced, semantic specification that applies to both subject and quality dependencies is that of the property that Thinklab adds to the underlying ontology to capture the semantics of the dependency. By property we refer to the ontological specification of the semantics for the connection between the two concepts. We have not discussed properties in these beginner-level modules, and this point can be safely skipped if it comes out as confusing. Otherwise, the specification uses the **for** keyword, like so:

```
...
observing
            (Elevation as measure
im.geography:Elevation in m) for
im.geography:hasElevation named elevation
```

This specification is necessary only when full control of the model semantics is desired. Thinklab will create a property automatically if one is not supplied, and in common modeling practice the specification is not required.

Quality dependencies

In quality models, dependencies are a fairly simple affair: in addition to the local concept, the observer and a name, an optional status can also be specified:

```
observing
      (Elevation as measure im.geography:Elevation
in m) optional named elevation
```

This allows Thinklab to compute the model even if that dependency cannot be resolved to a model (a **mandatory** keyword is also provided for completeness, but it's the default setting).

When an optional dependency cannot be resolved, the corresponding value will be **unknown.** In a deterministic calculation, this makes the result of any expression where it is used **unknown,** even if other values are valid. In aBayesian network (BN) model the BN will use prior probabilities instead of data; a result will be calculated but it will carry greater uncertainty than one where values can be resolved for all model dependencies. In the current Thinklab implementation there are some limitations to this behavior: for example multiplications do not yet allow unknown values. Such situations should be handled using the strategy explained in the expressions section below. This situation may arise even when the dependencies are resolved, e.g., when a dependency is resolved to data that contain "no data" values. So it's important that the expression code, or any other selected computation, is prepared to handle the resulting **unknown** value.

It is acceptable to specify the dependent observer and concept in an independent model as long as it is in the same namespace and defined before the model that uses it. For example:

```
namespace my.namespace using im, im.geography;

private model SoilPH as
      classify (rank im.geography:Soil
im.chemistry:PH) by im:Level into
            im:High if > 5,
            im:Low otherwise;

model SomethingDependentOnPH
      as ...
      observing
            SoilPH named soil-ph
      ... ;
```

can be written using the name of the model (SoilPH) in place of a full definition of it in parentheses. This is mostly useful when the same model needs to be reused in the dependencies of several models below it. When using this syntax, dependent models should always be declared **private**, to ensure that they are not inadvertently used to resolve a dependency in another namespace. In most cases, based on this example, the choice of whether pH is high or low should be decided in the context of the model using those subjective levels. If other models use the same conceptualization of high or low pH, the above approach will ensure that the same interpretation is used throughout (or simply to save some typing). In a collaborative context, however, subjective definitions should not be allowed to potentially "contaminate" other namespaces by being available to interpret soil pH, even if it would only be used if my.namespace:SoilPH was referenced in a dependency. For example, other people may search for the concept and (inappropriately) use it based the fact that the name matches a modeling need of theirs, when the more correct approach would be for each modeler to define their own private models (Note: There will be cases where it is entirely appropriate to reuse a model in diverse contexts. This is why it is so important that the applicable context of each model be accurately recorded).

This syntax can also be used to "force" the use of a specific model to resolve a dependency, instead of letting Thinklab resolve the concepts itself. This should be reserved for special situations, as doing so "wires" models together in a rigid way, deactivating much of the utility of semantic modeling. If it is deemed desirable to wire models together this way, a better way would be to refer to the models by name:

```
private model SoilPH named the-ph-we-want as
      classify (rank im.geography:Soil
im.chemistry:PH) by im:Level into
            im:High if > 5,
            im:Low otherwise;

model SomethingDependentOnPH
      as ...
      observing
            the-ph-we-want named soil-ph
      ... ;
```

This linkage is guaranteed to work even if there are other models for the same concept in the same namespace (e.g., assigned to different spatial locations). As shown above, the name of a model is specified using the **named** keyword that follows the observable. As per Thinklab naming conventions, lowercase, dash-separated names are used for such identifiers. If needed, a model from another namespace can be used as long as it has been imported in the**namespace** declaration at the top of the namespace (the list following the word "using"):

```
namespace my.namespace using im, im.geography, (the-
ph-we-want) from my.ph.models;

...

model SomethingDependentOnPH
      as ...
      observing
            the-ph-we-want named soil-ph
      ... ;
```

A list of symbols within parentheses can be used to identify imported identifiers (multiple identifiers can be included in that list, separated by commas). An asterisk (*) can be used to import all symbols from a namespace, although this can generate confusion and unexplained errors (e.g., symbol redefinition) when either namespace is changed, so this should not be done routinely. When no list of symbols is given, as we normally do with imported ontologies, the system merely records the dependency of the namespaces, and ensures that the dependency is read before the namespace that depends on it is parsed. Note that circular dependencies should be avoided: the system will not complain about them, but the results may be unpredictable and symbols or concepts that are expected to be defined may not look so when circular dependencies are defined.

Again, the direct use of models in dependencies is a special situation that shouldn't be the norm. In the normal case, when an observer is specified with the dependency, quite a bit of information can be embedded in the dependency statement. This is helpful to "localize" subjective interpretations to the model they're meant for. This comes in handy for example when using a subjective trait like im:Level, which is likely to only have a precise meaning within each computation. The example above, for instance, can be written in one single model:

```
model SomethingDependentOnPH
      as ...
      observing
            (SoilPH as
                  classify (rank im.geography:Soil
im.chemistry:PH) by im:Level into
                        im:High if > 5,
                        im:Low otherwise) named
soil-ph
      ... ;
```

This way, the SoilPH concept will only have the semantic scope of the model it's in, and will never be used outside of it or appear in searches. This is the safest way to specify dependencies on observables that are interpreted in ways that only apply to one or a few models in a namespace; this approach should thus be adopted as a default.

A special syntax can be used when defining a dependency - a classification **by** trait:

```
...
im.ecology:Forest by
im.conservation:DegradationLevel
...
```

Because the word **by** is reserved for classifications according to a specific subjective trait, the statement above can be used without ambiguity instead of a much more verbose **classify** statement. The output of this dependency will be one of the possible values of **im:DegradationLevel** (**im:Low**, **im:Medium**, ….) when matched to a model that defines that trait for a **im.ecology:Forest**. This alternative syntax may help make model specification as parsimonious and intuitive as possible.

Subject models and dependencies

Subject models instruct Thinklab to observe subjects within the context of interest. At this point it may be difficult to understand what a subject model does: indeed, subjects are created more than computed, so what are subject models for?

As suggested above, subject dependencies are only admitted in subject models, because there is a contextual relationship between the dependent and the observable that makes no sense for a quality. For example, temperature can be observed in the context of a region, based on the observation of the region, but the region to which a temperature refers cannot be observed based on a measure of temperature.

Because subject observations are direct and do not need observers, their use as dependencies is fairly simple. For example, the following statement:

```
model ...
      observing im.infrastructure:Road;
```

is all it takes for Thinklab to look up a model that annotates a source of subjects annotated with the Road concept, e.g., a vector file. will lookup data or models that can produce roads in the context, and if found, the roads will be created. Each road is a subject in itself, linked to the root subject.

Like with qualities, a property (**for** keyword) and an optional status for the dependency can be specified. Adding a name (**named**) is possible, but not useful, because the model will not refer to the "value" of a subject in any expression, and it's possible for zero subjects to be obtained as the result of the observation (e.g., roads can be observed in a region that has no roads, and a perfectly good observation of no roads can be obtained). Names for dependent subjects should be attributed, when appropriate, directly in the subject sources using attributes, as in the example we showed in the previous section.

The same syntax works with concepts describing events and processes, since each of these observers "stand alone" and are observed directly.

When the dependency is on a subject, process, or event, there is another, very useful syntax that enables complex agent-based models to be initialized in a very simple way. It is quite common for a model to want to create subjects where no subject sources are available, but where subject sources may provide a context for the subjects we want. For example, we may have subject sources for households (say a vector file with a point per household) but want a model containing agents for the families inside them. The use of the keyword **each**, which we have seen in subject models before, inside the dependencies allows to specify that:

```
model …
      observing im.demography:Family at each
im.demography:Household;
```

This dependency will first try to observe households in the context of observation; if any are found, Thinklab will extract the context (e.g. the location in space) for each of them, and create a Family at that same context. As with any regular subject, the family semantics will be used to define any further observation; for example, if we have told Thinklab that each family must have an income quality associated, then Thinklab will attempt to observe that income for each family generated (this is explained in more detail below). If the knowledge base contains models for a Family, the most appropriate for the context will be retrieved and used to initialize each family, which may also create qualities or subjects (e.g. individual members) in each Family subject. Backed by a knowledge base with contributions from many collaborating participants, the single line dependency above may build a whole simulated world.

Resolving dependencies vs. making observations in a context

By this point it may be evident that the modeling workflow we outlined in Module 1 (creating a subject then observing concepts of interest within it) is an exact equivalent of writing a model with all the concepts of interested as dependencies then observing that model in the context of the root subject. The only difference is that in the second case, a generic quality is observed without specifying its observer. For example, if the Elevation concept is modeled in a region, the output will be the best observation of Elevation there, whether in feet or meters. By stating a dependency with its observer, specific observation semantics are guaranteed. Obviously this is crucial if the resulting values need to be used in equations, so it would not be acceptable in such a model to write:

```
model ...
      observing im.geography:Elevation;
```

because there would be no control over the meaning of the numbers output

by such a model. The form above is, however, fine for subjects, because subjects are observed directly, so there is no interpretation for their values. As a result, Thinklab only allows the extremely simple syntax shown above when the concept specifies a subject.

Automatically resolved dependencies

In some cases, the ontologies (i.e., abstract knowledge) contain conditions for some observable's semantic coherence that require particular observations to be made. For example, the definition of a concept for a Watershed may include a restriction that a watershed must have a spatial configuration that allows a stream network to exist within it. This can be expressed in the ontologies using statements that look like:

```
thing Watershed is im.geography:Region
      requires StreamNetwork;
```

If such requirements are present, Thinklab will automatically try to make the necessary observations when a subject like a Watershed is observed. This is because everything in Thinklab is semantically explicit, and the statement above says that no watershed can exist if it does not have an observable stream network. By using semantic restrictions, many tasks that are usually done explicitly by modelers become automatic. When a concept like a Watershed is used as the inherent context of the observable (e.g. `Runoff of Watershed`), it is guaranteed that the Runoff model will be able to access the stream network. If the model is run at all, it will run in a Watershed, and a Watershed can only exist if a StreamNetwork can also be observed.

Actions linked to transitions

The examples so far have been fairly simple; however, complex operations used in other modeling systems can also be used within Thinklab, such as temporally explicit simulation specifying discrete differential equations for rates of change, or agent-based models.

All these functionalities are enabled in Thinklab by connecting actions to transitions. The full set of possible actions and transitions is both under active development and beyond the intended level of these modules. We will introduce the ideas here for completeness, though they are currently not fully functional and the final syntax may differ.

Module 5 describes how scale is represented in Thinklab, but we have already encountered examples where we assign spatial and/or temporal extents to subjects. Those extents may be represented in a way that implies multiple states (e.g., grid cells or polygons for space, time steps for time). In such cases, Thinklab will choose a computation sequence for these multiple states, and carry on the observation by generating scale transitions. The moment when the simulated time moves from t to t+1 is a time transition.

Dynamic simulation gets is power by attaching actions to time transitions. That is specified using a statement we have seen already:

```
model ....
      over time
            integrate population-size as
[population-size + birth - death],
            change land-use to im.landcover:Urban if
[population-size > 100];
```

Agent-based modeling is accomplished by providing syntax for dependent subjects to make changes and to access their own observed world from within expressions.

The 'over time/space/…' syntax allows a full specification of scale for each individual model. This allows a fully multi-scale system to be specified very simply. It works very nicely with subject models like so:

```
model AdministrativeRegion
      observing
            Household at each HouseholdLocation,
            Administration at each CapitalCity;


....

model Household
      ....
      over time (step="1 day")
            ....;

model AdministrativeRegion
      ...
      over time (step = "30 day")
            ...;
```

This example illustrates in a simple way how each model can specify a different temporal resolution, which will be automatically negotiated by Thinklab. Of course all these models are linked automatically through the resolution process. Thinklab may thus choose a different household model for only some of the households, say for example when the poverty level is above a threshold. These examples show how the investment in learning to "think semantically" pays off in simplicity. Because the meaning of all involved entities are specified fully and unambiguously, the software can wire components and data together properly, accomplishing difficult tasks that would normally fall to the modeler.

The `on definition` syntax previously seen in the section on data annotations is a special case of a transition: the difference is that when we say `on definition,` we refer to the initialization transition, which brings the model from the uninitialized to an initialized state, just before the temporal component of a simulation begins. The `on definition` syntax allows a list of actions just like `over ...`. However, some transitions that require the existence of time (e.g., `integrate`) are not allowed.

Bridging to external computations

Expressions in Thinklab are full Groovy programs, with which complex models and strategies can be coded. Yet, that requires programming skills, and deferring the logics of complex models to Groovy code is certainly not the primary reason for Thinklab to exist. In fact, many computations that are of interest to modelers are handled by external software, and when possible it is much simpler and cheaper to reuse external modeling software than to write new code to use inside Thinklab – particularly when the modeling software has an independent history and is maintained by others. For this reason, Thinklab provides a mechanism for a model to bridge to any external software that can be run from a language supported by the Thinklab implementation. This includes nearly all non-GUI based programs, as a modeling engine can easily integrate software and run external executables - but cannot as easily click buttons or fill forms.

The link between Thinklab and external computations is established using function syntax that follows the `using` keyword:

```
model SoilCarbonStored as
measure aries.carbon:SoilCarbonStored in t/ha
discretized by im:Level into
    im:VeryHigh if 200  to  520,
    im:High     if 110  to  200,
    im:Moderate if  90  to  110,
    im:Low      if  50  to   90,
    im:VeryLow  if 0.01 to   50,
    im:Minimal  if 0    to  0.01
observing
    im.geography:Slope by im:Level,
    im.soil:SoilPh by im:Level,
    ....
using bayesian(import="bn/madagascar/sink.xdsl");
```

The last line includes the `bayesian` function, which defines what we call a state accessor. This particular accessor is initialized by reading the file specified with the `import` argument from the project where the model is defined. The file specifies a Bayesian network which is then used to compute the value for each point in the context. The accessor will be passed the value of all dependencies and will compute the SoilCarbonStored result.

There are many accessors available in Thinklab in addition to the Bayesian one illustrated above. They include random number generators, table and spreadsheet readers, random choosers for outcomes based on probability distributions that can be parameterized using observed dependencies, and GIS operations. The existing ones are listed in the full documentation. Because function names are not keywords of the language, it is very easy for a developer to add new functions when needed, and the list of available accessors will keep growing with time.

Just as quality models can use the `using` syntax to pass off the computation of values to an external accessor, subject models can have subject accessors that operate on the subject as a whole, after all its quality dependencies have been initialized. For example, this is the current definition of the Watershed module included in the Thinklab hydrology ontology:

```
model im.hydrology:Watershed,
      //  pit-filled land elevation.
      (im.hydrology:Elevation as measure
im.hydrology:Elevation in m),
      (im.hydrology:FlowDirection as measure
im.hydrology:FlowDirection in degree_angle),
      (im.hydrology:TotalContributingArea as measure
im.hydrology:TotalContributingArea in m^2)
      ...
      observing
            (Elevation as measure
im.geography:ElevationSeaLevel in m)
      over space
      using hydrology.watershed();
```

The bulk of the hydrological computations is not specified as a giant expression at the end, but is left to the subject accessor named in the last line, which will be passed the full digital elevation map (guaranteed to be a map by the `over space` clause) and produce all the outputs indicated in the list. Subject accessors provide convenient ways to encapsulate complex computations in clean “packages” that can be written as Thinklab plug-ins in a variety of languages, providing unlimited points of extension associated to specific semantics while keeping the code clean and readable.

Multiple observables

A model doesn’t necessarily have only one output: indeed, most non-trivial models, such as the watershed model shown above or any Bayesian network with intermediate nodes, usually have more than one. If we want to keep results for future analysis beyond the “primary” observable that comes after the keyword `model`, we must provide observers for all the qualities that the model produces (subjects are directly observed, so they don’t require further specification). In the example above, we have used the same syntax that we used for dependencies to provide an observer for each additional output we want to keep. The accessor will be passed a list of the inputs and outputs, and will negotiate with the underlying model code to ensure that these can be produced and passed to Thinklab once computed, to become part of the final “dataset” represented by the subject.

The same specification can be used for anything computed by a state accessor. For example, a Bayesian network can be provided with observers for all the computed intermediate nodes and their relative uncertainties, using the `uncertainty` observer. The implementation of the accessor will ensure that the passed observers correspond to nodes in the network that can be computed, and will then create the corresponding observations.

Each of the observables in a model can be used to resolve dependencies within other models on its observable. All else being equal, Thinklab will try to minimize the number of models used, choosing a single model that produces two required observables over two models that produce the same observables independently, unless the latter are “evidence” (data) models.

Module 5. How to make model choices depend on context.

In previous modules, we have often hinted that “Thinklab chooses the best model” to observe concepts directly requested by the modeler or model dependencies. We have not, however, described in detail how that process occurs, or how to instruct Thinklab under what conditions a given model should be used. This section explains how models are chosen and how to control the model selection process.

Five fundamental topics will explain this process:

1. How to restrict the scope of a model to a specific scale (i.e., space or time; in this module we will only give examples about spatial regions);
2. How to use conditional statements to choose between different observers at each computed state;
3. How to use lookup tables to direct model selection when multiple methods exist to observe an observable;
4. How to use scenarios to “force” the use of certain models when particular model elements in an observation should reflect non-default assumptions;
5. How to tell Thinklab how much to trust a given dataset or model, which can become a factor in the resolution process.

Scale constraints for models and namespaces

One of the most useful things in modeling, particularly when we contribute to a shared model base, is to constrain a model for use only in a particular region. By region we refer to any constraints on the model's scale, so we could refer to a particular extent of space or time (e.g., falling within a certain polygon or temporal period), to a given resolution of either, or both. Thinklab uses a very general definition of scale, which extends to time, space, and even other "conceptual" dimensions, such as the hypothesis space (or multiple spaces) that is reflected in a model's assumptions. Those aspects are experimental and not discussed here, so in this documentation, we only describe how to set spatial constraints. The remaining aspects of scale and scale constraints will be covered in the full documentation.

The following examples describe spatial coverage and constraints, with limited discussion of temporal specifications. In general, anything that we say for space can be applied to time in a similar way, but the resolver may not be yet prepared to deal with all cases of temporal specification.

Constraining a model

When data, or an object source, are inherently spatial or temporal, it may not be necessary to specify constraints on the model's usage. In this case, their coverage is automatically recorded by Thinklab and becomes the "default" spatial or temporal constraint for all models that annotate it. For example, since the following data specification:

```
model wfs(urn =
"im:af.tz.landcover:tanzanialandcover",
            attribute = "lc")
named tanzania-lulc
as classify im.landcover:LandCoverType into
      im.landcover:AgriculturalArea        if "AG",
      im.landcover:ForestSeminaturalArea   if "NVT",
      im.landcover:VegetatedStillWaterBody if "NVW",
      im.landcover:UrbanFabric             if "UR",
      im.landcover:WaterBody               if "WAT";
```

specifies WFS access to a vector file for Tanzanian land cover, the bounding box reported by the WFS service for it will be used as its spatial constraint. Because of this, the `tanzania-lulc` model will only be used as a candidate for resolving the observable concept `im.landcover:LandCoverType` when the context of the request overlaps this bounding box. This is usually enough for raster spatial sources, whose coverage is exactly that rectangular bounding box – or should be ("no-data" borders are a different issue that can be dealt with using conditional statements, discussed later).

In some cases, it may be necessary to "correct" the coverage, either because 1) the data contain regions where they are unreliable, or 2) because (e.g., in a vector file covering a region that is far from rectangular) we want to ensure that other models will be used in regions covered by the bounding box of a preferred dataset, where we know that the first dataset is unavailable or unreliable. In such cases, we can use an `over space` keyword in a similar way as the `observe` statement:

```
model wcs(id = "san_pedro:swregap_lulc")
named vegetation-type-swregap
as classify aries.carbon:VegetationType into
      ...
over space (shape = "EPSG:4326 POLYGON((-114.816209
42.002018,..))");
```

(other ways to specify polygons aside from WKT include the use of shapefiles stored either 1) locally on the user's machine, 2) within a Thinklab project, or 3) on a GeoServer; complete descriptions and examples are provided in thefull documentation). Importantly, the spatial coverage specified after `over space` is intersected with the coverage of the namespace, when one has been given. That is, if a data source is used that does not cover the given polygon at all, the intersection will be empty and Thinklab will generate a warning message. Otherwise, the model will only be used in this example to resolve `aries.carbon:VegetationType` in the intersected spatial coverage. This can be useful when it is desirable to select only a specific portion of a larger coverage.

When not working with a data source (i.e., when annotating an unresolved model), nothing changes, except that there will be no native coverage with which to intersect. The `over space` notation can still be used, and the model will only be applied within the specified polygon.

A very common use of scale constraints occurs when annotating computed models that are meant to be used only in a specific region. This could either be a large range such as the tropical or temperate zone, or a smaller range where certain assumptions about a model's applicability can be considered valid. The cleanest way to do that is to constrain the whole namespace, adding additional constraints to individual models when necessary. The syntax for that is similar but uses the keyword `covering`:

```
namespace aries.carbon.local.sw-north-american-
deserts
      using im, im.hydrology
      covering space( shape = "....");
```

Because WKT specifications can be long and messy, a common strategy to keep the code clean is to use a specific namespace in a project to hold definitions for these locations:

```
namespace aries.carbon.locations;

define COASTAL_CALIFORNIA as
      space(shape = "EPSG:4326 POLYGON((-
122.01075303165209 38.46721456396898, ...))");

define MADAGASCAR as
      space(shape = "EPSG:4326
POLYGON((52.778320305152796 -27.644606378394307, ...
))");

define NORTHERN_ROCKIES as
      space(shape = "EPSG:4326 POLYGON((-111.05
45.01, -104 45.01, ...))");

define ONTARIO as
      space(shape = "EPSG:4326 POLYGON((-
95.35682310773775 50.520669204331895,...))");
```

then import the needed definitions into namespaces that need them, using defined identifiers to reference them:

```
namespace aries.carbon.local.northern-rockies
      using (NORTHERN_ROCKIES) from
aries.carbon.locations,im.soil, im, im.hydrology
      covering NORTHERN_ROCKIES;
```

The extended form of the `using` clause in the 'namespace' statement has been seen before, and can also be used to import symbols such as model names. In addition to improved readability, the 'using' clause has the advantage that definitions need be provided only in one place (i.e., per Thinklab project). If it is changed later (e.g., to a detailed polygon after testing it with a simple polygonal bounding box), it will automatically

affect all the namespaces that use it.

When a namespace is constrained to a particular region, all the models within it will be constrained to that region. If a model in a constrained namespace also incorporates an `over space` statement, Thinklab will intersect the namespace-level coverage with the model-specific one, further restricting its coverage, as seen before for data sources. So a namespace coverage can be restricted but not redefined on a model-by-model basis.

Temporal coverage

While we don't yet provide full support for temporal constraints, the `over time` statement should be used when appropriate to specify the time period covered by models or data. The most typical example is to identify the year to which a data source refers:

```
model ...
over time(start = 1995)
```

This indicates that the data are valid from 1995 onwards: they will not be used if the context has an `over time` definition that specifies an earlier year. An `end` year can also be specified, as can both a 'start' and an 'end' year. When faced with a choice of two models that are both temporally suitable to resolve a concept, Thinklab will give preference, all else being equal, to the most current one. If the context is temporal, this will be the one whose date is closest to the context's time; otherwise the model with the most recent start date will be used.

The `time` syntax is much more powerful than indicated here. It is possible, for instance, to specify full dates and times, resolutions etc., but as mentioned above, temporal support in Thinklab is still under development and full details will be provided in a future release.

Conditional choice of observer

As we learned earlier, any computations specified for quality models are carried on each state implied by the scale of the context. For example, a 10x10 spatial grid will be computed 100 times, once per cell, and the observer will be called upon to produce a value every time. So far we have seen models in the form:

```
      model <quality observable>
            as <observer> .... ;
```

Quality models may have more than one observer, which computes the value in different ways. We can thus assign conditions for choosing an observer, which may depend on other observations. These observers must be compatible, i.e., produce the same kind of observer/observation type. For example, measurements and proportions cannot be mixed, because that would break the model's semantic consistency. The general form for these conditional models is:

```
model <quality observable>
      [observing
            <model dependency> named <name>, ....]
      as
            ( <observer 1> ) [if <condition> ], ....
      ;
```

where the part in square brackets should be read as "optional." Each observer is in parentheses and may optionally be followed by the keyword `if` and an expression (using the square bracket notation). If the set of dependencies following the `observing` keyword is given before any observers are specified (i.e., before the `as` keyword), they will be computed before the observers are called in, and their value will become available for use in the expressions. The next example should clarify the syntax. In the (rather twisted!) model below, elevation and slope data are queried and the model will return values of zero, except where elevation is greater than 1000 m, where it will return the elevation as a value:

```
model CrazyElevation


/*
 * model dependencies - used only to select
observers..
 */
observing (Elevation as measure
im.geography:Elevation in m) named el


/*
 * two observers with conditionals. Parentheses are
not required in this
 * case but are good practice, as the condition for
the observer could be
 * wrongly attributed to the preceding observer's
action if the action
 * itself is unconditional.
 */
as
      (measure im.geography:Elevation in m
            observing
                  (Slope as measure
im.geography:Slope in degree_angle) named pslope1
            on definition
                  change to 0 if [pslope1 < 10] )
      if [el < 1000],


      (measure im.geography:Elevation in m
            observing
                  (Slope as measure
im.geography:Slope in degree_angle) named pslope2
            on definition
                  change to 0 if [pslope2 > 10] )
      otherwise;
```

(note that everything between `/*` and `*/` is interpreted as a comment, i.e., ignored by Thinklab). This example also shows how `otherwise` can be used as a catch-all condition instead of `if.` Both the `if` part and the model dependencies are optional; the behavior of this form when neither are supplied is very useful, because the "chain" of observers will be followed in the specified order until one of the observers produces a valid result. This is very useful to yield an alternative model when the preferred one produces `unknown` (no data) values, as it often happens when using spatial datasets. This form can also be used in resolved models. The only thing that cannot be done is to use incompatible types of observers as alternatives.

Lookup tables

While the conditional form shown above is useful, in some cases it will be easier and cleaner to just tabulate alternative values. Thinklab provides a powerful lookup table syntax, where the values in a column of the table can be returned on the basis of values in other columns.

Here is an example of a lookup table:

```
Landcover     Slope   Erosion factor
Rock          *       0.0
Sand          < 1     0.2
Grassland     < 1     0.04
Sand          1 to 4  0.4
Sand          4 to 7  0.6
Sand          > 7     0.8
```

In a model, it might be desirable to produce erosion factors that correspond to dependencies for land cover and slope. The standard statement to define a lookup table is similar to a function. It can be used directly in a model or as the argument of a **define** statement, to be referenced in other models and namespaces as shown earlier in the specification of spatial constraints. The previous definition can be stated as follows:

```
define EROSION_TABLE as table (landcover, slope,
erosion-factor):
      Rock,         *, 0.0,
      Sand,      < 1, 0.2,
      Grassland, < 1, 0.04,
      Sand,   1 to 4, 0.4,
      Sand,   4 to 7, 0.6,
      Sand,      > 7, 0.8;
```

and used in a model as follows:

```
model ErosionFactor as
      proportion ErosionFactor
      observing
            (LandCover as classify
im.landcover:LandCoverType) named land-cover,
            (Slope as measure im.geo:DegreeSlope in
degree_angle) named slope
      using lookup (land-cover, slope) into
EROSION_TABLE;
```

While the form above should be intuitive, there are several things to note. First of all, table definition can be set directly in the model if it is only needed there, by typing the ‘table’ and what follows instead of the EROSION_TABLE identifier:

```
   model ErosionFactor as
      proportion ErosionFactor
      observing
            (LandCover as classify
im.landcover:LandCoverType) named land-cover,
            (Slope as measure im.geo:DegreeSlope in
degree_angle) named slope
      using lookup (land-cover, slope) into table
(landcover, slope, erosion-factor):
                  Rock,         *, 0.0,
            Sand,      < 1, 0.2,
                  Grassland, < 1, 0.04,
                  Sand,   1 to 4, 0.4,
                  Sand,   4 to 7, 0.6,
                  Sand,      > 7, 0.8;
```

The choice of which syntax to use is only one of convenience, and should be dictated by the need to reuse the table elsewhere. In general, inline (latter, model-embedded) specifications should be used unless the table is “official” (e.g., reflects accepted standards) or would need to be reused through the code.

The rows of the table can contain simple values to be matched, but it is also possible to use a classifier specified with the **classify** statement as a table entry. Each dependency will be matched following the order of the column list indicated after the keyword ‘lookup.’ In the previous example, the land-cover value for each point within the context will be matched to values in the first column and the slope to values in the second. The result of the lookup operation will yield values in the last column for the first row that matches both classifiers.

The lookup values to be computed as output can be associated with a column other than the last one by using the **?** identifier in the **lookup** call. For example, the call above is equivalent to **lookup (land-cover, slope, ?)** and the **?** could be used in any position (i.e., land-cover, ?, erosion-factor). This way, a lookup table can be used with greater flexibility.

A ***** classifier will match any value. Be careful when using it - it should always be the last choice within each set of otherwise equivalent combinations, which can be ambiguous when there are several columns. When in doubt, remember that choices are always matched top to bottom.

The names chosen for the columns after the **table** keyword do not influence the way the lookup table works: the list is only used to define the _number of elements required for each row. This is crucial to the proper functioning of the table, as Thinklab does not rely on or mandate indentation and formatting. As with any component of a semantic modeling system, however, it is important that descriptive, unambiguous names are used for column headings, so that the meaning of the table is clear to anyone reading it.

Lookup tables can be even more powerful because each classifier or value can also be an expression. Expressions are normally used to compute the values to be returned; in such cases, they will be computed before the value is assigned, and these computations can use all the model dependencies. If they are matched instead, the match will be successful only when the dependency associated with the column and the result of evaluating the expression is the same.

The choice of whether to use a lookup table or a conditional statement (described earlier in this module) depends on the context. One approach or the other may be the cleaner method depending on the model, its purposes, and the modeler’s preferred coding style.

Scenarios

Scenarios in Thinklab are sets of alternative models used only when the scenario is explicitly activated. When one or more scenarios are active, the models within them will always be chosen preferentially to resolve their concepts. Conversely, models in scenarios that are not active will never be used. For example, a climate change scenario may contain alternative datasets for temperature, precipitation, or other climatic variables. Activating this scenario will guarantee that any observation of precipitation and temperature will reflect the scenario’s assumptions. As a concept can describe observables at any level in the model dependency chain, applying a scenario can affect an entire modeling session or just limited elements of it.

Specifying a scenario in Thinklab is as simple as creating a namespace using the keyword **scenario** instead of **namespace** in the first statement.

```
scenario aries.ipcc.scenarios.hadley.b2
     using im.geography;

model wcs(id = "usa:sum_hi_wint_lo_hadley_B2")
      named summer-high-winter-low-hadley-b2-north-
america
      as measure im.geo:SummerHighWinterLow in
Celsius;

...
```

Scenarios are activated explicitly by the modeler (in the Thinkcap GUI this is done by ticking the appropriate checkbox for the scenario in the “Scenarios” view) before observing the concept(s) of interest. When

observations are made with any scenarios active, any dependency associated with the concepts modeled in the scenarios will then be resolved using the scenario instead of the "regular" knowledge base. If, for example, a scenario contains a land cover change model that observes the im.landcover:LandCoverType concept, any model that depends on im.landcover:LandCoverType will link to that land cover change model, attempting to resolve it using data or models from the scenario namespace, before resorting to the standard data layers. No models or data included in scenarios will ever be used unless the scenario is active.

Some scenarios are inherently incompatible with others; for example, different climate change scenarios should not be mixed together, because they reflect different assumptions about emissions trajectories. Other scenarios could be appropriate for combined use. For example, it might be appropriate to run a scenario for climate change and one for land cover change individually and then in combination to explore their synergistic effects. When scenarios are mutually exclusive, as in the case of multiple climate or land-cover change scenarios, the **disjoint with** clause can be added to the scenario specification to ensure that the listed scenarios are mutually exclusive:

```
scenario aries.ipcc.scenarios.hadley.b2
      disjoint with aries.ipcc.scenarios.hadley.a2,
aries.ipcc.scenarios.hadley.b1
                  using im.geography;

...
```

As explained, the navigator in the modeler interface we provide (Thinkcap) has a "Scenarios" view with checkboxes that allows their activation. Since we defined the IPCC scenarios as disjoint, ticking the checkbox for one of them will automatically deselect the others. Scenarios not declared disjoint can be used together without restrictions.

Influencing the model ranking: subjective metrics of quality

Thinklab uses a fairly sophisticated ranking algorithm to select which model to use when more than one model is found that could observe an observable. We do not give full details on the algorithm here, though it is important to list the criteria that it uses. One of these criteria can be influenced by the user-supplied metadata for each model, reflecting the modeler's "rating" of e.g., data quality or model reliability, so we will explain how to use this feature. Modelers can influence the ranking algorithm in much deeper ways, but that's an advanced (and potentially dangerous) topic that we will not discuss here.

Thinklab currently uses the following criteria to rank models. Criteria are listed in the default order of importance that Thinklab gives them when computing the rank of a model (note that this is a very active area of development, so the criteria, ordering, or definition may change):

1. **lexical scope** reflects whether the model is in a scenario, in the same namespace of the dependency that makes the observation, or in the same project; models that are located "closer" to where they are needed are given preference. Models that are in active scenarios are always chosen above all others. Otherwise, for instance, a model in the same namespace as one that requires its observable will be preferentially chosen.
2. **trait concordance** reflects the number of attributes (traits) that the candidate model's observable shares with the observable to be resolved. Attributes "percolate" through a model chain starting with the context. So if we are modeling in a im.geography:Region that has been tagged with an attribute (e.g., im:May), models that share that attribute (e.g., data that refer to the month of May) will be chosen preferentially.
3. **scale coverage** reflects how much of the scale defined in the selected context is covered by the model.
4. **scale specificity** reflects the ratio between the total coverage of the candidate model vs. that of the context. Models that are more specific will be prioritized over models that have been constrained to larger contexts or have not been constrained at all. For example, a regional-scale model (dataset) would typically be selected ahead of a national-scale model, which would be selected ahead of a global-scale model.
5. **inherency**: models that are specifically meant to be observed in the particular type of context being used will be chosen preferentially over more general models.
6. **subjective concordance**: this criterion uses a multiple-criteria ranking of user-defined metadata vs. a weight structure that can be redefined on a namespace-by-namespace basis (see below).
7. **evidence**: resolved models with data sources will be chosen preferentially vs. computed models.

Aside from the choice to activate or deactivate scenarios, number 6 is the only criterion that is under direct user control, i.e., the "subjective concordance" criterion. These can be defined by users, but the current convention in Thinklab uses only one criterion on a routine basis, named "im:reliability." Such criteria are specified in metadata at the end of a model statement (before the final semicolon), like so:

```
      model
            ..... (full model definition)
      with metadata {
            dc:originator "NCAR GIS Climate Change
Scenarios"
            dc:url "http://www.gisclimatechange.org"
            im:reliability 75
            im:distribution "public"}
      ;
```

Metadata specification is fairly flexible, and any metadata tag or value could theoretically be used without generating a syntax error (though a consistently defined and applied set of metadata conventions is of course highly desirable in a collaborative modeling environment). For these criteria, we use the convention of specifying values using positive integers between 1 and 100. The default intermediate value for any criterion that is evaluated but not given in metadata is 50. So each model will have im:reliability = 50 unless the modeler enters a different value. Unless the default ranking priorities are changed (see below), user-specified reliability will then be used as the value to assess the above-defined criterion 6. If a model is thought to be of particularly poor quality (e.g., coarse resolution, minimally documented, or with other known limitations), it should receive a lower value; models of high quality should receive a higher one. Conventionally we have preferred using the 25-75 range, leaving extreme values for special situations, though for certain well-known, methodologically robust data or models higher values (e.g., 90) may be warranted.

Thinklab provides a vocabulary for other criteria, including for example "openness" that may be used to nudge the model choice towards those that are open source. The current version, however, only uses im:reliability.

Each namespace can redefine the entire ranking strategy using the following syntax:

```
namespace my.namespace
      resolve using {
            im:lexical-scope 1
            im:evidence 3
            im:trait-concordance 2
            im:subjective-concordance 4
            im:scale-coverage 5
            im:scale-specificity 6
            im:inherency 7
            im:scale-coherency 8
            im:network-remoteness 0
            im:reliability 100
      };
```

where each criterion name not corresponding to one of the “core” criteria (1-7 above) [CLARIFY] is matched to the metadata using the indicated weight. **Use of this form is very dangerous unless the implications of doing so are well understood**. If multiple subjective criteria are present, they will be aggregated using a multiplicative weighted multiple criteria algorithm that we do not discuss here. The modified ranking strategy will be used to resolve any model included in the namespace for which the modified ranking has been created.

Lastly, “blacklist” and “whitelist” namespaces can be added for use in model resolution by using the following syntax:

```
      namespace picky.namespace1
            resolve from
                  good.namespace1,
                  good.namespace2

      namespace picky.namespace2
            resolve outside
                  bad.namespace1,
                  bad.namespace2;
```

The blacklist (`resolve outside`…) and whitelist (`resolve from`…) are not needed together, as the whitelist will select only those namespaces for resolution. It will effectively ignore the blacklist, which tells Thinklab to avoidresolving from blacklisted namespaces. Conversely, using only a blacklist would eliminate models in blacklisted namespaces from use, making a whitelist unnecessary. When ranking instructions are provided together with a black/white list, `resolve` is only used once:

```
      namespace my.namespace
            resolve from
                  good.namespace1,
                  good.namespace2
            using {
                  im:lexical-scope 1
                  im:evidence 3
                  im:trait-concordance 2
                  im:subjective-concordance 4
                  im:scale-coverage 5
                  im:scale-specificity 6
                  im:inherency 7
                  im:scale-coherency 8
                  im:network-remoteness 0
                  im:reliability 100
            };
```

Using such specification can give provide power and flexibility to over the way in which a model is resolved. However, it is also likely to lead to situations that are confusing and difficult to manage unless great care is taken, so we suggest that they be avoided by all but expert users. # Reference sheets

This part of the documentation contains quick reference sheets that can be useful during modeling practice.

The unit reference details the syntax for units of measurement understood by Thinklab’s unit parser. The function reference briefly describes the most common functions in Thinklab and their arguments. The glossary lists the terms used most often in Thinklab documentation and provides a brief definition for each of them.

This section is far from complete and cannot substitute person-to-person instruction yet. # Unit of measurement: reference chart # Thinklab functions: reference chart # Glossary of terms used in Thinklab/ARIES modeling

Abstract knowledge: Concepts; abstract knowledge provide a general definition of each concept and is contained within ontologies.

Accessor: A function that links Thinklab’s core code and functionality to an external method or program for data and model handling. The accessor transfers inputs from Thinklab to the external program and returns outputs to Thinklab for analysis by the modeler.

Action: A change in the state of a model element that may occur, for example, during a transition.

Annotation property: [UPDATE DEFINITION]

Bayesian network: A probabilistic graphical model (a type of statistical model) that represents a set of random variables and their conditional dependencies via a directed acyclic graph.

Bayes’ Theorem: A mathematical theorem that allows estimation of the likelihood of one event that is conditional on another. It allows the probabilities of linked events to be updated from a state where information about the likelihood of events is lacking (and only prior probabilities exist) to a state where an outcome is known (evidence of an event is submitted and posterior probabilities are estimated via Bayesian updating).

Bayesian updating: The process by which prior and/or conditional probabilities are replaced by evidence (knowledge of the state of an event), and are updated using Bayes’ Theorem to become posterior probabilities.

Child node: A child node in a Bayesian network is influenced by the state of one or more parent nodes via an edge (arrow) that indicates influence. Child nodes may include intermediate nodes whose outcomes determine the state of additional child nodes and top nodes whose states do not influence other nodes, and are typically a final output of a Bayesian network model.

Computed observation: An observation that has been calculated deterministically, i.e., using equations.

Concept: The description of an entity [UPDATE DEFINITION]

Conditional probability: The probability of outcomes in a child node, which depend on the states of all possible combinations of values for the parent nodes. Conditional probabilities are updated to posterior probabilities once evidence is submitted, during the process of Bayesian updating, and can also be updated by training the Bayesian network.

Context: The conditions under which a model may be run (in Thinklab terms, conditions under which an observable may be observed). In Thinklab this will most commonly be a set of geographic and/or temporal constraints under which a model may be run.

Data model: A model statement that directly references a specific dataset to be observed, as opposed to a model that requires Thinklab to search for other means to resolve the concept.

Deterministic model: A model where every set of outcomes is determined by predefined model parameters and input data states, and identical results are obtained every time when input data values are the same.

Event: An observable that has both a subject and a temporal component. [UPDATE DEFINITION]

Extensive physical property: When measuring an extensive property, the amount of the substance being measured (or more frequently in Thinklab, the area over which it is measured) greatly matter. Mass and volume are examples of extensive physical properties – so when aggregating e.g., tons of biomass or volumes of water, the extent of the analysis must be very carefully considered.

Extent: In Thinklab, a given subset of time and/or space that can be used to generate the scale under which a model can be run.

Identity: [UPDATE DEFINITION]

Indirect observation: Qualities produce indirect observations: since a quality cannot be observed without an associated subject, an observation of a quality with an associated subject is an indirect observation. [UPDATE DEFINITION]

Inherency: The association of specific qualities or properties with a given subject.

Intensive physical property: When measuring an intensive property, the amount of the substance being measured (or more frequently in Thinklab, the area over which it is measured) does not matter. Temperature or density are examples of intensive physical properties.

Knowledge graph: A graphical display in Thinkcap that shows the relationship between a selected concept and all other related concepts (i.e., within an ontology).

Literal: [UPDATE DEFINITION]

Mediation: A method for defining different states that could be observed for a model, typically used with classifications, in a way that links an observable to traits and inherencies to produce more readable, compact modeling statements.

Model knowledge: Models; each piece of model knowledge describes a potential way to resolve a concept (i.e., through data or models). Models may: 1) directly reference and annotate a piece of data (which is itself simply a way to observe an observable) or 2) include equations, algorithms, or external processes to compute an observable, which may include dependencies on additional observables to compute the state of the model's observable.

Namespace: Each project in Thinkcap can contain multiple namespaces – individual files that contain abstract and model knowledge. A namespace can be thought of as an individual file within a folder (i.e., Thinkcap project).

Object: [UPDATE DEFINITION]

Observable: A subject, process, quality, or event that could be viewed and quantified using a model.

Observation: The viewing and quantification of a concept within a given context. A modeling process may yield multiple observations (i.e., instances) of the concept within a given context. Multiple methods may also exist to observe an observable (i.e., various data and models).

Observer: Thinklab uses seven observer types, which are included in an observer statement for each model. Observer types include rankings, measurements, counts, values, classifications, proportions, and ratios. Selection of the appropriate observer type for each data or model is critical.

Ontology: A file that contains abstract knowledge of concepts. This includes definitions for concepts, spatial and temporal constraints on concepts, and relationships between concepts. Ontologies in Thinklab are organized by general thematic areas (e.g., hydrology, landcover, soils).

Parent node: A parent node in a Bayesian network influences the state of one or more child nodes via an edge (arrow) that indicates influence. The state of the parent node is, however, not influenced by the state of any other node.

Posterior probability: The probability of outcomes following Bayesian updating, which occurs once evidence is submitted for one or more nodes in the Bayesian network. Following the updating process, posterior probabilities thus replace prior and conditional probabilities.

Prior probability: The probability of the occurrence of different states of a parent node, prior to the submittal of evidence on the Bayesian network.

Probabilistic model: A model where input data are defined using probability distributions rather than constant values. Monte Carlo simulation and Bayesian modeling are two examples of common probabilistic modeling approaches.

Process: A process always has an inherent subject. [UPDATE DEFINITION]

Project: A Thinklab project can contain multiple namespaces, as well as other files. Thinklab currently contains several core projects (im and org.aries), a tutorial project (thinklab.tutorial), and individual projects that contain data and models for various ecosystem services (org.aries.carbon, org.aries.sediment, org.aries.water, etc.). Additionally other projects may be created as testing and development spaces.

Property: [UPDATE DEFINITION]

Quality: The result of a process; qualities are indirect observations of phenomena and cannot "stand alone" in observations. For example, the sediment load of a river would be a quality (the sediment load is the quality and cannot be independently observed without a subject, in this case the river).

Raster data: A cell-based configuration for spatial data, where the extent covered by the data includes a grid of cells at a specified spatial resolution (e.g., 25x25 m), and each cell carries a certain value (e.g., for elevation, land cover, or precipitation).

Reification and **De-reification**: [UPDATE DEFINITION]

Resolution: The process by which Thinklab applies search algorithms to iteratively match concepts to models as many times as necessary, and uses heuristics and artificial intelligence to define the most suitable models at each step. Once the appropriate model(s) are selected, they will be run and the appropriate observations will be passed back to the modeler. Successful resolution of the model yields a resolved observation.

Restriction: Within ontologies, a restriction specifies a "requirement" for a concept that links together related concepts. [CLARIFY]

Scale: In Thinklab, the spatial and/or temporal constraints on a model, i.e., conditions under which it can be considered valid to run a given model to generate observations.

Semantic modeling: A paradigm that maintains the meaning of all entities being modeled through the use of ontologies that provide clear, consistent definitions of concepts and the relationships between concepts (i.e., entities that can be modeled, and data and models to observe them).

State: Successful observation of a concept will produce a state for each observation. Individual observations of this state will be distributed across the spatial and temporal scale set by the context of analysis.

Structural learning: An advanced form of Bayesian training where an algorithm determines the structure of the Bayesian network (i.e., dependencies between parent and child nodes) based on the data, as well as their associated probabilities. Structural learning is a complex process and is not planned for integration with ARIES/Thinklab in the near future.

Subject: The thing to be observed during the modeling process. Subjects may also have associated qualities or properties, some of which may be inherent to a specific subject.

Thinkcap: A graphical user interface (GUI)-based client run within the Eclipse software development environment. Thinkcap communicates with the Thinklab server, which parses and runs the Thinklab modeling language.

Thinklab: A semantic modeling language and the system that parses and runs it (i.e., the Thinklab server).

Training: The application of an algorithm, e.g., expectation maximization, to learn and quantify the relationships between the nodes of a Bayesian network based on data for at least one parent and one child node within the network. Training replaces user-supplied conditional probabilities with those derived from the data.

Trait: A trait provides further descriptive information about an observable, yielding semantic precision while keeping the size of Thinklab's core ontologies tractable. Traits can be used, for example, to provide finer temporal specification or to specify levels or frequencies of the observable.

Transition: A change from one state to another. For example, in a temporally dynamic model, a transition marks the movement of a simulation from one time step to the next, and is associated with an action that is coded into the model for a given transition.

Uninformed prior: A state of total absence of knowledge about a system, in which the likelihood of each of several states are equal (i.e., given four possible states, each would be assumed to have a probability of occurrence of 0.25).

Vector data: A non-cell based configuration for spatial data. Data may take the form of polygons, lines, or points. Each of these features will have one or more associated attributes.

Thinklab naming conventions

Supplemental material

This section contains material that relates to the practice of modeling with Thinklab but is not directly related to the language or its implementation.

At the moment the only content available is a primer on Bayesian networks, given the common usage of these methods in models developed with Thinklab.

We also added the beginning of a Thinklab cookbook to show some examples of commonly used expressions and programming patterns. This section will be expanded with time. # Using Bayesian networks for ecosystem service modeling in ARIES

Bayesian network models (BNs, also called Bayesian belief networks) are a class of probabilistic models that can aid in quantifying ecosystem services – particularly their source, sink, and use conditions. BNs have been applied to a wide variety of research problems across the sciences, and others have used them in ecosystem services assessments, but ARIES was one of the first tools to explicitly incorporate BNs. However, it is critical to note that ARIES is amodeling platform that integrates multiple modeling paradigms – both probabilistic and deterministic. BNs are but one approach to quantifying ecosystem services and like any modeling approach have their strengths, weaknesses, and contexts where their use will be more or less appropriate. ARIES therefore integrates probabilistic OR deterministic models as appropriate to the context of interest to the modeler. However, since most modelers are more familiar with deterministic modeling approaches, we are providing this module to familiarize new users with Bayesian modeling approaches.

In mathematical terms BNs are a probabilistic graphical model (a type of statistical model) that represents a set of random variables and their conditional dependencies via a directed acyclic graph. A BN specifies a joint distribution in a structured form; dependence between the variables is represented by a directed graph. Random variables are represented by nodes, and edges indicated direct dependence. "Directed" refers to the fact that certain factors influence others (via direct correlation or causal influence) in a directional way. One or more parent nodes thus influence each child node in a BN model. Each parent and child node carries with it a set of probabilities that it will take a certain value or state. For parent nodes this is termed the prior probability and for child nodes it is termed the conditional probability (i.e., the probability of its value is conditional on a set of outcomes for its parent nodes). The BN is acyclic because it cannot account for feedback loops in the way that some types of deterministic models do. BNs use Bayes' Theorem to update the probability (yielding posterior probabilities of the distribution of values for a child node when new evidence (i.e., data) is submitted for the state of its parent node(s), and vice versa).

There are at least three important benefits associated with Bayesian modeling:

1. Bayesian models explicitly account for and communicate uncertainty in their results. However, they only handle uncertainty related to missing data; it is not possible for them to account for uncertainty associated with the underlying model structure (with the exception of advanced approaches such as structural learning that are not planned for immediate integration into Thinklab so will not be discussed in depth here).

2. Bayesian models can operate under conditions where deterministic models do not exist or are known to perform poorly. For example, the Revised Universal Soil Loss Equation (RUSLE) was developed to quantify and map soil erosion. It has been tested and performs particularly well on relatively level slopes and in agricultural systems. It performs more poorly on steep slopes (>20%), on geologically young, mountainous soils, and in quantifying rill, gully, and streambank erosion. In ecosystems where ecological production functions are poorly known, or in social systems where demand for an ecosystem service is minimally understood, Bayesian networks may provide a superior approach for ecosystem service mapping and quantification. A productive way forward within ARIES could thus be to combine known, well-tested deterministic models (e.g., RUSLE) in locations where their performance is well accepted, and to pair these with probabilistic (Bayesian) approaches elsewhere.

3. Bayesian models can operate under conditions where data are missing or incomplete. For example, if an input spatial dataset

for a region of interest covers only half of that region, the model will use the data for the half of the region where it exists, and will use the prior probabilities (further discussed below) for the part of the region where no data exist. Results will be generated for the entire region, though they will have less uncertainty where data exist and greater uncertainty where data do not.

Below, we present general guidelines for developing BNs, drawn from an excellent article by Marcot et al. (2006). We do not, however, present a step-by-step guide to using a particular BN editing software. A variety of BN editing software platforms are available, including both commercial software packages such as Netica (https://www.norsys.com/), Agenarisk (http://www.agenarisk.com/) and freeware, such as GeNIe (http://genie.sis.pitt.edu/index.php/downloads); we encourage the reader to consult the help documents and tutorials supplied by these software providers.

Guidelines for Bayesian modeling (following Marcot et al. 2006)

Construction, testing, and use of Bayesian models entails six steps:

1. Develop the causal graph
2. Discretize each node
3. Assign prior probabilities
4. Assign conditional probabilities
5. Peer review
6. Test with data and train the Bayesian network

1. Develop the causal graph

The first step in BN construction is to develop what can be called a causal graph, influence diagram, or “alpha-level model.” Through consulting with experts or the literature, the modeler gains an understanding of what inputs or variables influence a system’s behavior. These include parent nodes whose values influence the likelihood of a distribution of outcomes for one or more child nodes (intermediate nodes may also be used to aggregate the influence of multiple parent nodes, themselves influencing an ultimate “top” child node). Influence is indicated by an arrow pointing from the parent to the intermediate or top node, and indicates a causal relationship; unlinked nodes are assumed to be unrelated.

Best practices for developing causal models include:

1. Keeping the number of parent nodes and the number of their discrete states (more on this to follow) low enough that conditional probability tables (CPTs) remain tractable. As a rule of thumb, Marcot et al. suggest limiting each child node to no more than 3 parent nodes and each parent node to no more than 5 discrete states (this is where intermediate nodes can be valuable in keeping CPTs tractable). The number of combinations required to complete a CPT will be the number of states of each parent node multiplied by each other. So if a model has three parent nodes, with two having two discrete states and the third having five discrete states, 20 different probability combinations will need to be entered in the CPT. It is thus easy to see how CPTs can become intractable when too many parent nodes and states are added to the model. As always, model simplicity (i.e., parsimony) is generally a desirable goal!
2. Ensuring that each node corresponds to an observable, i.e., a unique, semantically specified thing, quality, process, or event. Typically these will correspond to spatial data. In some cases intermediate nodes can be used to aggregate parent nodes in “fuzzier” concepts that help to keep BN CPTs tractable, but overreliance on this approach is undesirable. In cases where data for an observable do not exist but data for a related proxy do, an additional parent node can be added to correspond to the proxy dataset, and an appropriate degree of uncertainty can be added to the intermediate node being approximated by that proxy dataset.
3. Not developing models that are overly “deep,” i.e., contain too many layers of nodes (Marcot et al. suggest using four or fewer layers). Too deep a model often indicates an undesirably high level of complexity; additionally, the effects of submitted evidence (data) for parent nodes on the state of the top child node is watered down by the presence of intermediate nodes, so the model may become unresponsive to changes in input data.
4. Always documenting models, particularly the chosen rationale for model construction choices. This may involve citations of the literature or a list of experts consulted on model structure. See Bagstad et al. (2011) for example documentation of past ARIES models.
5. BNs should be constructed at the appropriate scale. In ARIES, BNs have been constructed to quantify source, sink, or use values or inputs to those final values; however a different class of flow models are used to link source, sink, and use regions, so sources, sinks, and uses should typically not be mixed within the same BN.

2. Discretize each node

Once the causal model is complete, every node must be classified into a set of discrete states. While some BN modeling platforms support the use of continuous data nodes, GeNIe (the BN modeling software used to date with ARIES) does not, so discretization is a necessary next step. Very often, these will be related to traits (e.g., present/absent or high/moderate/low, which is often appropriate for discretizing continuous data), though they could also be categorical (e.g., soils group or land cover type). It is important to strike the right balance between precision, over-simplification, or over-complication of the model. As a general rule, if a different value of the input data would produce a different outcome, those input data states should take separate discrete states in the BN. When multiple states of the input data are expected to yield the same results, these states could be combined into a single discrete state in the BN.

In addition to defining the number and names of each discrete state in the BN, the modeler must also define how data for the appropriate observable will yield values for each possible discrete state. For instance, continuous data must be discretized to as many non-overlapping ranges as there are discrete states for the data. Discretized categorical data may entail a list of unique categories if the modeler expects each category to yield different results or a combination of categories (i.e., into multiple groupings of soil, vegetation, or land cover types) if (s)he expects all members of each group to produce similar model behavior. If certain data values exist for the region of interest but they are not included in the discretization, ARIES will assume there are no data for those locations. It is thus important that the entire range of values be included within the discretization. Generally, information from past ecological production function studies is the best way to discretize data. When such studies are absent, as is often the case, it may be sensible to explore the data for the modeler's region of interest in GIS and to use an algorithm (e.g., Jenks natural breaks, equal interval) to discretize continuous data.

Accurate discretization of data for the top node of a model is extremely important. The possible values for discrete states of the top node should correspond to those the modeler expects to observe in their region of interest where the model will run. Particularly if training data are unavailable at the needed spatial and temporal resolution, knowing the possible range of outputs in the context of interest is critical. If done correctly, high and low model output values will thus be realistic for the context of interest. When adequate training data are absent, data ranges can be derived from coarser resolution data, published field or modeling studies, or expert elicitation.

3. Assign prior probabilities

For parent nodes, prior probabilities must next be assigned. Prior probabilities are the expected likelihood of a particular discrete state occurring in the absence of data. Obviously, both prior and conditional probabilities must sum to 1 across all the discrete states. Three general approaches can be used to set prior probabilities – 1) use existing data (i.e., look at the data for a similar region of interest and determine the frequency at which each discrete state occurs), 2) use expert elicitation, or in the absence of either, 3) use uninformed priors. The latter are simply 1 divided by the number of discrete states (i.e., if there were 5 discrete states, each would be set to 0.2). Uninformed priors represent a total absence of knowledge about the system. Remember, when input data exist for all or part of a region of interest, the BN will be set to the corresponding discrete state from the data for that location; priors will be used only where data do not exist (i.e., for incomplete or patchy datasets).

4. Assign conditional probabilities

Conditional probabilities define the probability distribution of the states of a child node for each potential combination of parent states. As discussed previously, a large number of parent nodes and/or discrete states for those nodes will result in very complex CPTs, increasing the burden on the modeler to generate well documented, defensible contingent probabilities.

If an equation exists that defines the relationship between parent and child nodes, it is possible to use it to calculate values for the child node CPT. Otherwise, when using expert elicitation, a common approach is to "peg the corners," defining the highest and lowest potential combinations of values from the parent nodes. These cases are set to 0% and 100%, and the modeler gradually interpolates values for the intermediate cases.

Copying and pasting probabilities to a spreadsheet program can often be a helpful approach to completing CPTs. The modeler can sum the rows for each combination of parent node states, ensuring that all values sum to 1 - a necessary condition for all prior and contingent probabilities, based on the definition of a probability distribution of a discrete random variable. If the modeler knows that a certain parent node is more influential in determining the value of the child node, that information should be represented in the CPT. For instance, imagine a 3-node BN where we must complete a CPT for a child node with two parents, each parent having 3 discrete states (yielding a 9-column CPT). The modeler could start by pegging the corners for just the most influential parent node, completing a simple 3-column CPT. (S)he could then determine how much the less influential parent node influences the value of the child node, and adjust the values for the six additional columns in the final, 9-column CPT.

In a few cases, it may be appropriate to simplify complex CPTs using a noisy max algorithm. Rather than allowing every possible combination of values of the parent nodes to exist (which can quickly result in very complex CPTs), noisy max nodes simplify the CPT by considering only the possible states for each individual parent node – not their interactions. We have used noisy max nodes, for example, in viewshed and open space proximity models where objects in a field of view or nearby open space types are either mutually exclusive or the most dominant characteristic determines the outcome for the child node, rather than a combination of the interacting values from multiple parent nodes. For instance, in a model of visual blight we assume that for each cell where the model is run, a highway, commercial development, or a transmission line will each degrade a viewshed to a certain degree individually, rather than in combination. The "leak" column specifies the likelihood that each discrete state of the child node would occur when all parent node conditions are absent.

Once the CPTs are populated the "alpha-level" model has been completed. The modeler should test the BN, setting evidence for the different input nodes and updating the model to ensure that the intended behavior is expressed within it. Changes in evidence for more influential relationships should yield greater changes to the model's output, and vice versa. If unexpected behavior occurs, model structure and/or probabilities should be adjusted.

5. Peer review

To generate a "beta-level model," the network structure, prior and contingent probabilities, and model behavior should be reviewed by a subject matter expert who was not involved in the initial construction of the BN. This person can either agree with or recommend changes to all or part of the BN, yielding a more robust model.

6. Test with data and train the Bayesian network

The final step in BN construction and testing – generation of the final "gamma-level model" – entails training the model to actual data. Assuming we have data that correspond to multiple parent and child nodes, the process of BN training uses an algorithm, e.g., expectation maximization, to "learn" the relationships between the nodes, replacing the CPTs, which are often subjective or expert-driven, with probabilities derived directly from data. This is the true "data-driven modeling" approach that BNs bring to probabilistic modeling. Some more complex algorithms derive the structure of the BN as well as the values of the CPT for a given set of input data (i.e., structural learning). However, this is a complex process and Thinklab support for structural learning is not planned for the near future.

While BN training is an extremely desirable last step in model development and testing, it will not always be possible. Even if datasets that correspond to the concepts expressed by all or most parent and child nodes in the BN exist, they must be of adequate quality for the training to succeed. Data that vary too widely in their temporal currency will be more likely to introduce error into a trained BN, since different layers will represent conditions at different times. Another frequently encountered problem occurs when data occur at varying spatial resolutions. When one or more coarse resolution datasets are trained along with one or more fine resolution datasets, many different values from the fine resolution dataset may be matched to a single value from the coarse resolution dataset. For example, if a 25x25 m and a 1 km^2 dataset are used to train a BN, 160 different possible inputs from the 25x25 m dataset may be matched to a single value of the 1 km^2 dataset. This can yield a trained BN that may be quite insensitive to changes in its input data. In such cases, an untrained but well-vetted and documented BN may be preferable to a BN trained using poor quality data.

Parting thoughts

As previously noted, BNs are not a panacea for ecosystem service modeling (nor is any single modeling approach or paradigm). BNs can, however, be very useful particularly in places where data are scarce, trusted deterministic models do not exist, or such models are known to perform poorly. When the opposite conditions hold, deterministic models are likely to outperform probabilistic models; as always, a best practice is for the modeler to know his/her system and to carefully choose the most appropriate data and models for that system. Once they are semantically annotated, ARIES is equally capable of handling probabilistic and deterministic models, and the ARIES resolver will be able to select the most appropriate model for the modeling problem of interest.

References

Bagstad, K.J., F. Villa, G.W. Johnson, and B. Voigt. 2011. Artificial Intelligence for Ecosystem Services (ARIES): A guide to models and data, version 1.0. ARIES report series n.1.

Marcot, B.G., J.D. Stevenson, G.D. Sutherland, and R.K. McCann. 2006. Guidelines for developing and updating Bayesian belief networks applied to ecological modeling and conservation. Canadian Journal of Forest Research 36: 3063-3074.

McCann, R.K., B.G. Marcot, and R. Ellis. 2006. Bayesian belief networks: application in ecology and natural resource management. Canadian Journal of Forest Research 36: 3053-3062.

Pearl, J. 1988. Probabilistic reasoning in intelligent systems: Networks of plausible inference. Morgan-Kaufmann: San Francisco.

Vigerstol, K.L. and J.E. Aukema. 2011. A comparison of tools for modeling freshwater ecosystem services. Journal of Environmental Management 92(10):2403-2409.

Yudkowski, E.S. 2003. An intuitive explanation of Bayes' Theorem. http://yudkowsky.net/rational/bayes

Thinklab cookbook

This section is meant to support a "learn by example" paradigm for task of common occurrence. These examples are mostly meant to show expressions and their use, and use simple semantics that should not be taken as indicative of good semantic modelin practices. In semantic modeling, more than anywhere else, it is fundamental to use examples to learn, not to cut and paste from - the practice is bad everywhere, but particularly so in a paradigm that is founded on what things mean: cutting and pasting is the opposite of thinking!

Computing states

The examples shown below illustrate how to build analytic (i.e., algebraic) models using the Thinklab language. The keys to this are two clauses:

```
on definition set to [some formula of the dependent concepts]
```

```
on definition change to [some formula of the dependent concepts]
```

The "set to" clause assigns the value to be a function of the dependent concepts referenced in the observing clause. The "change to" clause uses the dependent concepts to modify the state of the top-level observable.

Importantly, if "change to" is used, the system will expect a previous value for the observable to be defined at the time the expression is evaluated, and will try to resolve it using data or other models; the formula may contain the model name to refer to the unmodified observable. If "set to" is used instead, the model is expected to produce the value directly and will work as long as all the dependencies are resolved, and the use of the model name in the formula is an error.

Note that within the square brackets [] any valid Groovy code may be evaluated, provided that it returns a value that matches the model type. For more information on the Groovy language, see http://groovy.codehaus.org

To model a concept as an analytical function of the values of one or more other concepts:

```
model newConcept as measure newConcept in
```

Example using the above model statement to express the equation: TotalCarbonStorage = VegetationCarbonStorage + SoilCarbonStorage

```
model TotalCarbonStorage as measure
aries.carbon:TotalCarbonStorage in t/ha observing
(VegetationCarbonStorage as measure
aries.carbon:VegetationCarbonStorage in t/ha) named
vegetation-c, (SoilCarbonStorage as measure
aries.carbon:SoilCarbonStorage in t/ha) named soil-c
on definition set to [vegetation-c + soil-c];
```

Another example which expresses the equation: GreenhouseGasEmissions = PopulationDensity * 0.04

```
model GreenhouseGasEmissions as measure
im.policy:GreenhouseGasEmissions in t/ha*year
observing (PopulationDensity as count
im.policy:Population per km^2) named population-
density on definition set to [population-density *
0.04];
```

Another example which expresses the equation: YieldUse = PopulationDensity * YieldUseCoefficient

```
model YieldUse as measure ia.agriculture:YieldUse in
kg/year observing (ia.agriculture:PopulationDensity as
count im.policy:Population per km^2) named population-
density, (YieldUseCoefficient as ratio
YieldUsePerPerson) named yield-use-coefficient on
definition set to [population-density * yield-use-
coefficient];
```

Another example which expresses the equation: LiveStockWaterUse = CowDensity * WaterUseCoefficientCattle + SheepDensity * WaterUseCoefficientSheep + GoatDensity * WaterUseCoefficientGoat

```
model LivestockWaterUse as measure
aries.water:LivestockWaterUse in mm observing
(CowDensity as count im.agriculture:Cattle per km^2)
optional named cow-density, (SheepDensity as count
im.agriculture:Sheep per km^2) optional named sheep-
density, (GoatDensity as count im.agriculture:Goat per
km^2) optional named goat-density,
(WaterUseCoefficientCattle as ratio
aries.water:WaterUsePerHeadOfCattle) optional named
water-use-coefficient-cattle,
(WaterUseCoefficientSheep as ratio
aries.water:WaterUsePerSheep) optional named water-
use-coefficient-sheep, (WaterUseCoefficientGoat as
ratio aries.water:WaterUsePerGoat) optional named
water-use-coefficient-goat on definition set to [(cow-
density * water-use-coefficient-cattle) + (sheep-
density * water-use-coefficient-sheep) + (goat-density
* water-use-coefficient-goat)];
```

A tricky example which expresses the piecewise equation: IrrigationWaterUse = {2000 if LandCover = AgriculturalArea, 0 otherwise} Note the use of “change to” rather than “set to” in this model statement.

```
model 2000 named default-irrigation-amount as measure
aries.water:IrrigationWaterUse in mm observing
(LandCover as classify im.landcover:LandCoverType)
named land-cover on definition change to 0 unless
[land-cover == im.landcover:AgriculturalArea];
```

An example using the Normal distribution. This expresses the piecewise equation: Elevation = {0 if Normal(mean=542.0,std=223.3) < 500, Normal(mean=542.0,std=223.3) otherwise} Note the use of “change to” rather than “set to” in this model statement.

```
model rand.normal(mean = 542.0, std = 223.3) named
random-elevation-filtered as measure im.geo:Elevation
in m on definition change to 0 if [random-elevation-
filtered < 500];
```

An example using the Poisson distribution. This expresses the piecewise equation: ResidentialWaterUse = {0 if Poisson(lambda=12) <= 22, Poisson(lambda=12) otherwise} Note the use of “change to” rather than “set to” in this model statement.

```
model rand.poisson(lambda = 12) named random-
scattered-residential-users as measure
aries.water:ResidentialWaterUse in mm on definition
change to 0 if [random-scattered-residential-users <=
22];
```

An example using the || operator to merge a River and Spring layer together. WaterPresence = true if RiverPresence or SpringPresence is true

```
model WaterPresence as presence of
im.hydrology:WaterBody observing (RiverPresence as
presence of im.hydrology:River) named stream,
(SpringPresence as presence of im.hydrology:Spring)
named spring // “presence of” is a boolean value, so
we use the OR (||) operator here, true if either or
both are true. on definition set to [stream ||
spring];
```

An example using a rank (unitless quantification) observation type. WildlifeSpeciesRichness = (AmphibianRichness + BirdRichness + MammalRichness + ReptileRichness) * 0.25

```
model WildlifeSpeciesRichness as rank
aries.recreation:WildlifeSpeciesRichness observing
(im.ecology:AmphibianRichness as rank
im.ecology:AmphibianRichness) named amphibian-
richness, (im.ecology:BirdRichness as rank
im.ecology:BirdRichness) named bird-richness,
(im.ecology:MammalRichness as rank
im.ecology:MammalRichness) named mammal-richness,
(im.ecology:ReptileRichness as rank
im.ecology:ReptileRichness) named reptile-richness on
definition set to [(amphibian-richness + bird-richness
+ mammal-richness + reptile-richness) * 0.25];
```

CODE ONLY

Creating a context

```
EPSG:4326 POLYGON((-70.8783850983603 -3.3045881369117316,-69.05465462961
-3.2661991868835875,-69.13155892648547 -4.575963434877864,-
70.91683724679801 -4.630717874946029,-71.15853646554768
-4.263784638927346,-70.8783850983603 -3.3045881369117316))"
observe im.geography:Region leticia-peru over space(
    grid = "500 m",
    shape = "EPSG:4326 POLYGON((-70.8783850983603 -3.3045881369117316,-
69.05465462961 -3.2661991868835875,-69.13155892648547
-4.575963434877864,-70.91683724679801 -4.630717874946029,-
71.15853646554768 -4.263784638927346,-70.8783850983603
-3.3045881369117316))"
);
observe im.hydrology:Watershed leticia-peru over space(
    grid = "500 m",
    shape = …
);
```

Module 2. Models as observations: subjects, qualities and traits.
Concepts and observables
The primary observable
Keeping ontologies simple

```
observe im:Annual im.hydrology:Watershed
    over space(...)
model ... as measure im.climate:Rainfall in mm;
model .... as measure im:Annual im:Average im.climate:Rainfall in mm;
model Elevation as classify (measure im.geography:Elevation in m) by
im:Level into
    im:Low if 0 to 350,
    im:Medium if 350 to 1000,
    im:High if 1000 to 8000;
```

Identities managed by authorities

```
<abstract observable> identified as “<key>” by <authority>
count im.ecology:Individual identified as "5212442" by GBIF
```

Inherent qualities and subjects

```
model ... as measure im.geography:Elevation within im.geography:Region in
m;
```

Module 3. Connecting data to models: semantic annotation and observation semantics.
Choosing a concept
Choosing the data or subject source
Values

```
model 100 as measure im.chemistry:Water im.physics:BoilingTemperature in
Celsius;
model false as presence of im.theology:Satan;
model im:High as classify (probability of im.climate:ClimateChange) by
im:Level;
```

Data sources

```
<function-name> ( <argument-name> = “parameter_value”, …)
model wcs(urn=”im:global.geography:dem90m”, no-data = -32768.0) as
measure im.geography:Elevation in m;

model raster(file="data/landcover.tif") as
    classify im.landcover:LandCoverType into
        im.landcover:Urban        if 200,
        im.landcover:Agricultural if 201,
        ....
        ;
model wfs(urn="im:af.tz.landcover:tanzanialandcover", attribute="luc_id")
as
    classify im.landcover:LandCoverType into
        im:Agricultural if "AG",
```

```
        ....
        ;
```

Subject sources

```
model each wfs(urn = "im:global.infrastructure:global_rail_merge")
named railroad-global
as im.infrastructure:Railway;
```

Observation semantics for qualities

```
model ....
    as ....
    using
        (Elevation as measure im.geography:Elevation in m) named el,
        ....
        ... instructions to compute the result based on 'el';
```

Ranking

```
model wcs(...) as rank ...;

model wcs(...) as rank ...:PerceivedDanger 1 to 5;
```

Measurement

```
measure ...
model wcs(urn = "im:na.us.climate.annual:annualprecip")
    named precipitation-annual-2007-usa
    as measure im:Annual im.hydrology:PrecipitationVolume in mm
    over time (year = 2007)
    on definition change to [precipitation-annual-2007-usa * 0.01];
```

Count

```
model 1 as count Universe;
model wcs(urn="aries:global-populationdensity-2006") as
    count im.demography:HumanIndividual per km^2
    over time (year = 2006);
```

Value

```
model ... as value of ...:PropertyParcel in USD@2004;
```

Classification

Direct classification

```
model ManureType as
    classify im.agriculture:Manure into PigManure, CattleManure,
PoultryManure
  observing
        (PigManureProportion as proportion of im.agriculture:Pig  in
im.agriculture:Manure im.core:Mass) named pig-manure,
        (CattleManureProportion as proportion of im.agriculture:Cattle in
im.agriculture:Manure im.core:Mass) named cattle-manure,
        (PoultryManureProportion as proportion of im.agriculture:Poultry
in im.agriculture:Manure im.core:Mass) named poultry-manure
    using rand.select(
        distribution = (pig-manure cattle-manure poultry-manure),
        values = (PigManure CattleManure PoultryManure)
    );
```

Indirect classification

```
model data.wcs(id = "europe:corine2000", no-data = 255) named corine-2000
    as classify im.landcover:LandCoverType according to im:numeric-
encoding;
...
class LandCoverType
    has children
        ...
        (DiscontinuousUrbanFabric with metadata { im:numeric-encoding 112
})),
        (class IndustrialCommercialTransport
            has children
                (class IndustrialCommercialUnits with metadata
{ im:numeric-encoding 121 }),
                (class RoadRailNetwork with metadata { im:numeric-
encoding 122 }),
```

```
                (class PortArea with metadata { im:numeric-encoding
123 }),
        ...
```

Classifying values into observables or traits

```
model wfs(urn = "im:af.tz.landcover:tanzanialandcover",
        attribute = "lc")
named tanzania-lulc
as classify im.landcover:LandCoverType into
    im.landcover:AgriculturalArea        if "AG",
    im.landcover:ForestSeminaturalArea   if "NVT",
    im.landcover:VegetatedStillWaterBody if "NVW",
    im.landcover:UrbanFabric             if "UR",
    im.landcover:WaterBody               if "WAT";
model wcs(...) as classify im.policy:Poverty of
im.demography:HumanPopulation by im:Level into
        im:High     if 4,
        im:Moderate if 3,
        im:Low      if 2,
        im:Minimal  if 1;
model wcs(...elevation data...) as measure im.geography:Elevation in m
    discretized by im:Level into
        im:High if > 2000,
        ....                        (An INCORRECT example)
namespace my.namespace using im.geography;

(RIGHT, BUT STILL NOT PERFECT)

private model Elevation as
    classify im.geography:Elevation by im:Level
        im:High if > 2000,
        ....;
```

Proportion and percentage

```
model ... as percentage of im.agriculture:Pig  in im.agriculture:Manure
im.core:Mass;
model … as proportion im.ecology:CanopyCover;
```

Ratio

```
model … as ratio of im.ecology:Soil im.chemistry:Carbon im.core:Mass to
im.ecology:Soil im.chemistry:Nitrogen im.core:Mass;
```

Presence

```
model … as presence of im.infrastructure:Building;
```

Probability and uncertainty

```
model … as probability of im.physics:Fire within im.ecology:Forest;
```

Discretization

```
model wcs(...) as measure im:Length of im.ecology:Leaf in cm
    discretized by im:Level as
        im:Low    if < 10,
        im:Medium if 10 to 30,
        im:High   if > 30;
```

Note: spatial densities and temporal rates refer to observations, not observables

```
model .... as count im.demography:HumanIndividual per km^2;
```

De-reification of subject models

```
model each wfs(urn = "im:global.infrastructure:grand_reservoirs_v1_1")
named reservoirs-global
as im.hydrology:Reservoir
interpret
    GRAND_ID as im:name,
    AREA_SKM as measure im.core:Area in km^2;
model wfs(urn = “im:global.infrastructure:grand_reservoirs_v1_1”,
attr=”AREA_SKM”)
    named reservoirs-area-global
    as measure im.core:Area within im.hydrology:Reservoir in km^2;
```

Module 4. Computing deterministic and probabilistic observations.
Model syntax: observable, dependencies and computations.

```
namespace my.namespace using im, im.agriculture;

...

model SummerCropYield as
    measure im:Summer im.core:Yield of im.agriculture:Crop in t/km^2
    observing
        (JunCropYield as measure im:June im.core:Yield of
im.agriculture:Crop in t/km^2) named jun-yield,
        (JulCropYield as measure im:July im.core:Yield of
im.agriculture:Crop in t/km^2) named jul-yield,
        (AugCropYield as measure im:August im.core:Yield of
im.agriculture:Crop in t/km^2) named aug-yield
    on definition set to [jun-yield + jul-yield + aug-yield];
```

The context of applicability for a model

```
model SummerCropYield as
    measure im:Summer im.core:Yield of im.agriculture:Crop in t/km^2
    observing
        (JunCropYield as measure im:June im.core:Yield of
im.agriculture:Crop in t/km^2) named jun-yield,
        (JulCropYield as measure im:July im.core:Yield of
im.agriculture:Crop in t/km^2) named jul-yield,
        (AugCropYield as measure im:August im.core:Yield of
im.agriculture:Crop in t/km^2) named aug-yield
    over space
    on definition set to [jun-yield + jul-yield + aug-yield];
```

The observable in computed models

Mediation

```
namespace my.namespace;

...

model ElevationLevel as
    classify (measure im.geography:Elevation in m) by im:Level into
        im:High if > 1000,
        im:Low  if < 1000;
model ElevationLevel as
    classify ElevationLevel by im:Trait into im:High, im:Low
    observing
        (Elevation as measure im.geography:Elevation in m) named
elevation
    on definition set to [
        elevation < 1000 ? im:Low : im:High
    ];
```

Expression language

Using expressions in data models

```
 model wcs(...)
    named ghg-emissions-usa
    as measure im.policy:GreenhouseGasEmissions in t/ha*year
    on definition change to [ghg-emissions-usa * 0.0001];
```

Limitations

Dependencies in detail

```
…
observing
        (Elevation as measure im.geography:Elevation in m) for
im.geography:hasElevation named elevation
```

Quality dependencies

```
observing
    (Elevation as measure im.geography:Elevation in m) optional named
elevation
namespace my.namespace using im, im.geography;

private model SoilPH as
    classify (rank im.geography:Soil im.chemistry:PH) by im:Level into
        im:High if > 5,
        im:Low otherwise;

model SomethingDependentOnPH
```

```
    as ...
    observing
        SoilPH named soil-ph
    ... ;
private model SoilPH named the-ph-we-want as
    classify (rank im.geography:Soil im.chemistry:PH) by im:Level into
        im:High if > 5,
        im:Low otherwise;

model SomethingDependentOnPH
    as ...
    observing
        the-ph-we-want named soil-ph
    ... ;
namespace my.namespace using im, im.geography, (the-ph-we-want) from
my.ph.models;

...

model SomethingDependentOnPH
    as ...
    observing
        the-ph-we-want named soil-ph
    ... ;
model SomethingDependentOnPH
    as ...
    observing
        (SoilPH as
            classify (rank im.geography:Soil im.chemistry:PH) by im:Level
into
                im:High if > 5,
                im:Low otherwise) named soil-ph
    ... ;
...
im.ecology:Forest by im.conservation:DegradationLevel
...
```

Subject models and dependencies

```
model ...
    observing im.infrastructure:Road;
model …
    observing im.demography:Family at each im.demography:Household;
```

Resolving dependencies vs. making observations in a context

```
model ...
    observing im.geography:Elevation;
```

Automatically resolved dependencies

```
    thing Watershed is im.geography:Region
        requires StreamNetwork;
```

Actions linked to transitions

```
model ....
    over time
        integrate population-size as [population-size + birth - death],
        change land-use to im.landcover:Urban if [population-size > 100];
model AdministrativeRegion
    observing
        Household at each HouseholdLocation,
        Administration at each CapitalCity;

....

model Household
    ....
    over time (step="1 day")
        ....;

model AdministrativeRegion
    ...
    over time (step = "30 day")
        ...;
```

Bridging to external computations

```
model SoilCarbonStored as
measure aries.carbon:SoilCarbonStored in t/ha
discretized by im:Level into
   im:VeryHigh if 200  to  520,
   im:High     if 110  to  200,
   im:Moderate if  90  to  110,
   im:Low      if  50  to   90,
   im:VeryLow  if 0.01 to   50,
   im:Minimal  if 0    to  0.01
observing
   im.geography:Slope by im:Level,
   im.soil:SoilPh by im:Level,
   ....
using bayesian(import="bn/madagascar/sink.xdsl");
model im.hydrology:Watershed,
    //  pit-filled land elevation.
    (im.hydrology:Elevation as measure im.hydrology:Elevation in m),
    (im.hydrology:FlowDirection as measure im.hydrology:FlowDirection in
degree_angle),
    (im.hydrology:TotalContributingArea as measure
im.hydrology:TotalContributingArea in m^2)
    ...
    observing
        (Elevation as measure im.geography:ElevationSeaLevel in m)
    over space
    using hydrology.watershed();
```

Multiple observables

Module 5. How to make model choices depend on context.

Scale constraints for models and namespaces

Constraining a model

```
model wfs(urn = "im:af.tz.landcover:tanzanialandcover",
        attribute = "lc")
named tanzania-lulc
as classify im.landcover:LandCoverType into
    im.landcover:AgriculturalArea        if "AG",
    im.landcover:ForestSeminaturalArea   if "NVT",
    im.landcover:VegetatedStillWaterBody if "NVW",
    im.landcover:UrbanFabric             if "UR",
    im.landcover:WaterBody               if "WAT";
model wcs(id = "san_pedro:swregap_lulc")
named vegetation-type-swregap
as classify aries.carbon:VegetationType into
    ...
over space (shape = "EPSG:4326 POLYGON((-114.816209 42.002018,..))");
namespace aries.carbon.local.sw-north-american-deserts
    using im, im.hydrology
    covering space( shape = "....");
namespace aries.carbon.locations;

define COASTAL_CALIFORNIA as
    space(shape = "EPSG:4326 POLYGON((-122.01075303165209
38.46721456396898, ...))");

define MADAGASCAR as
    space(shape = "EPSG:4326 POLYGON((52.778320305152796
-27.644606378394307, ... ))");

define NORTHERN_ROCKIES as
    space(shape = "EPSG:4326 POLYGON((-111.05 45.01, -104 45.01, ...))");

define ONTARIO as
    space(shape = "EPSG:4326 POLYGON((-95.35682310773775
50.520669204331895,...))");
namespace aries.carbon.local.northern-rockies
    using (NORTHERN_ROCKIES) from aries.carbon.locations,im.soil, im,
im.hydrology
    covering NORTHERN_ROCKIES;
```

Temporal coverage

```
model ...
over time(start = 1995)
```

Conditional choice of observer

```
    model <quality observable>
        as <observer> .... ;
model <quality observable>
    [observing
        <model dependency> named <name>, ....]
    as
        ( <observer 1> ) [if <condition> ], ....
    ;
model CrazyElevation

/*
 * model dependencies - used only to select observers..
 */
observing (Elevation as measure im.geography:Elevation in m) named el

/*
 * two observers with conditionals. Parentheses are not required in this
 * case but are good practice, as the condition for the observer could be
 * wrongly attributed to the preceding observer's action if the action
 * itself is unconditional.
 */
as
    (measure im.geography:Elevation in m
        observing
            (Slope as measure im.geography:Slope in degree_angle) named
pslope1
        on definition
            change to 0 if [pslope1 < 10] )
    if [el < 1000],

    (measure im.geography:Elevation in m
        observing
            (Slope as measure im.geography:Slope in degree_angle) named
pslope2
        on definition
            change to 0 if [pslope2 > 10] )
    otherwise;
```

Lookup tables

```
define EROSION_TABLE as table (landcover, slope, erosion-factor):
    Rock,        *, 0.0,
    Sand,      < 1, 0.2,
    Grassland, < 1, 0.04,
    Sand,   1 to 4, 0.4,
    Sand,   4 to 7, 0.6,
    Sand,      > 7, 0.8;
model ErosionFactor as
    proportion ErosionFactor
    observing
        (LandCover as classify im.landcover:LandCoverType) named land-
cover,
        (Slope as measure im.geo:DegreeSlope in degree_angle) named slope
    using lookup (land-cover, slope) into EROSION_TABLE;
  model ErosionFactor as
    proportion ErosionFactor
    observing
        (LandCover as classify im.landcover:LandCoverType) named land-
cover,
        (Slope as measure im.geo:DegreeSlope in degree_angle) named slope
    using lookup (land-cover, slope) into table (landcover, slope,
erosion-factor):
            Rock,        *, 0.0,
        Sand,      < 1, 0.2,
            Grassland, < 1, 0.04,
            Sand,   1 to 4, 0.4,
            Sand,   4 to 7, 0.6,
            Sand,      > 7, 0.8;
```

Scenarios

```
scenario aries.ipcc.scenarios.hadley.b2
   using im.geography;

model wcs(id = "usa:sum_hi_wint_lo_hadley_B2")
    named summer-high-winter-low-hadley-b2-north-america
```

```
    as measure im.geo:SummerHighWinterLow in Celsius;

...
scenario aries.ipcc.scenarios.hadley.b2
    disjoint with aries.ipcc.scenarios.hadley.a2,
aries.ipcc.scenarios.hadley.b1
            using im.geography;

...
```

Influencing the model ranking: subjective metrics of quality

```
    model
        ..... (full model definition)
    with metadata {
        dc:originator "NCAR GIS Climate Change Scenarios"
        dc:url "http://www.gisclimatechange.org"
        im:reliability 75
        im:distribution "public"}
    ;
namespace my.namespace
    resolve using {
        im:lexical-scope 1
        im:evidence 3
        im:trait-concordance 2
        im:subjective-concordance 4
        im:scale-coverage 5
        im:scale-specificity 6
        im:inherency 7
        im:scale-coherency 8
        im:network-remoteness 0
        im:reliability 100
    };
    namespace picky.namespace1
        resolve from
            good.namespace1,
            good.namespace2

    namespace picky.namespace2
        resolve outside
            bad.namespace1,
            bad.namespace2;
    namespace my.namespace
        resolve from
            good.namespace1,
            good.namespace2
        using {
            im:lexical-scope 1
            im:evidence 3
            im:trait-concordance 2
            im:subjective-concordance 4
            im:scale-coverage 5
            im:scale-specificity 6
            im:inherency 7
            im:scale-coherency 8
            im:network-remoteness 0
            im:reliability 100
        };
```