An Introduction to ACUITy and the APVF Ontology[†]

Amy Aragones
Jeanette Bruno
Andrew Crapo
Marc Garbiras

General Electric Global Research
1 Research Circle
Niskayuna, NY 12309

October, 2005

[†] This work was partially funded by the Air Force Research Laboratory, Wright Patterson Airforce Base, under contract F33615-03-2-6300.

Table of Contents

Table of Figures

1 Introduction

This paper introduces the Adaptive Work-Centered User Interface Technology (ACUITy) software architecture, which embodies two novel concepts. The first concept is to use an ontology modeling approach to characterize a work domain in terms of "work-centered" activities as well as the computation mechanisms that achieve an implementation that supports those activities. The second and equally important concept is to provide adaptive interaction, both user directed and automated, in the work-centered characterization and presentation mechanisms of the user interface to decision support applications.

First introduced by Eggleston and Whitaker (2000), the goal of a Work-Centered Support System (WCSS) is to "provide an integrated and tailored support system that…offers support to work in a flexible and adaptable manner" by customizing user interaction according to the situated context in which work is accomplished. Under a work-centered approach, we develop an understanding of the overall targeted work domain: what the work domain encompasses, what the goals of work are, who participates in the work domain, and how the participants achieve the goals of the work domain, given local context. We use this understanding of the work domain to characterize and thus support what the participants are trying to accomplish in their day-to-day activities.

We extend Eggleston and Whitaker's approach by taking advantage of recent developments in semantic web technology to achieve a new level of adaptive decision support. In the ACUITy architecture, we explicitly characterize domain concepts and relationships in a hierarchy of ontologies, associated with upper level ontological constructs that enable adaptive reasoning and extensibility. Our core ontology is derived from three work-centered design principles introduced by Eggleston and Whitaker (2002), including:

1. *The Problem-Vantage-Frame Principle*: Effective decision support interfaces should display information that represents the perspective that the user requires on the situated work domain to solve particular types of problems.

2. *The Focus-Periphery Organization Principle*: Information that is the most critical to the user in the current work context should be displayed in the focal area of a decision support display to engage the user's attention; referential information should be offered in the periphery of the display to preserve context and support work management.

3. *The First Person Perspective Principle*: The user's own work ontology (terms and meaning) should be the primary source for information elements in the interface display.

Our modeling approach extends Eggleston and Whitaker by formalizing and simplifying the task of building and extending a work-centered support system. We also offer adaptive user interface capabilities through the use of a controller that can "reason" about

metadata in the ontology to present users with a work-centered application tailored to individual needs and responsive to changes in the work domain.

In this report, we provide an overview of the ACUITy architecture, focusing on key concepts in the ontology design and controller interactions. Please note that the ACUITy architecture, including the controller and ontologies, is a work-in-progress. We continue to extend, refine and reorganize the ontologies as we learn more about the work domains of interest, develop new capabilities and extend our understanding of ontology development and application. We will certainly need to pass through a productionization phase in order to transition this technology into use. However, our research has reached the point where it will benefit from wider discussion, feedback and, hopefully, development contributions. Our goal here is to support those activities by documenting the current state of our technology and future directions, mindful that there are still many open questions to be answered.

1.1 The Rationale for Adaptive Work-Centered Support Systems in Autonomic Logistics

At this time, our work domain of interest spans maintenance planning and logistics in both commercial and military applications. Improved productivity in field maintenance and supply is increasingly driven by automated, proactive information flow and planning. To that end, both military and commercial logistics operations are implementing monitoring, diagnostic and prognostic systems to increase the quality and accessibility of decision support information, including equipment status, resource and asset availability, appropriate repair strategies and other information required to make informed decisions related to maintenance support. This potentially results in an increased challenge to decision-makers to cognitively process, analyze and take action on a much larger amount of information.

Decision support systems for field maintenance and supply are typically focused on improving availability and productivity. In commercial applications, this translates into profitability by reducing costs and delivering on reliability, availability and performance requirements. In military applications, an efficient logistics operation facilitates mission readiness, reduces downtime and reduces the risk of lean sustainment. Although different operational policies, procedures and unknowns come into play in commercial and military contexts, the core problem is the same.

Maintenance planning is knowledge-based work that is subject to dynamic and unpredictable forces in logistics support systems. It is characterized by:

- Large amounts of distributed and heterogeneous information that must be accessed and comprehended in a timely manner
- Many business rules and other factors that constrain decision processes
- Geographically distributed collaboration in planning and problem-solving
- The need to analyze what-if scenarios to evaluate potential courses of action

- The need to prioritize work at the plan or project level as well as at the task level.

In this situation, it is easy for information overload, miscommunication, and loss of focus to occur, particularly in systems where the user typically utilizes multiple tools that are not seamlessly integrated or interfaced. Even in using systems that automate the fusion of data into information, users may become confused and cognitively overloaded if the process is inaccessible and untraceable. Information may be overlooked or misinterpreted, resulting in sub-optimal decisions.

In particular, autonomic logistics support systems capitalize on the information flowing from diagnostic and prognostic information systems to intelligently respond to a given situation with the correct repair plan and resources in the right amounts, at the right moment, at the right place [AFRL/HESS, 2002]. A cognitive agent architecture enables decisions to be made – and actions to be taken – autonomously, which realizes productivity gains not only from optimizing operational decisions relative to the state of the overall logistics system, but also from enabling decision-makers to efficiently focus on the work that requires their attention. A necessary component of this system is an adaptive interface that provides advanced decision support for human decision makers by bundling, prioritizing and presenting the information they need in order to deal with open-ended tasks. An adaptive interface paradigm shifts the design focus away from models of specific tasks with specific objectives and specific outcomes towards context-sensitive interfaces that respond to dynamic forces within the system.

The objective of the ACUITy research program is to develop adaptive and context-sensitive human-computer interfaces to support maintenance decision processes. There are two broad approaches generally taken to the problem of designing for adaptability:

- The interface could tailor itself to user needs based on dynamic, automated user models updated through observation of situated user action

- The interface could support adaptation by the user in response to unforeseen events or highly variable work domains that are not perfectly observable by computer artifacts.

We view these paradigms as highly complementary in the field maintenance work domain. The architecture described here responds to both the need for automated dynamic information composition and delivery as well as interaction capabilities that help the user to respond flexibly to dynamic problem-solving events.

1.2 Adaptive Work-Centered Support

Work-centered support systems are built upon the premise that a system that provides the user with the right information at the right time and in the right format will provide a real and substantial benefit. What constitutes the right information, the right format, and the right time will almost always depend on the user's objective and will often depend upon the user himself. These information needs are often dynamic, particularly in relatively open-ended work domains such as maintenance planning where equipment status and resource availability are uncertain. Since the WCSS is tasked with predicting what will be

most helpful to the user in a given context, WCSS software in this domain will necessarily be model-based. Predictive models of work in general and the particular work domain in specific must exist before the relevance of information can be evaluated. Models of user behavior in general, of users playing specific roles in this work domain, and of a user's personal preferences are also necessary to enable tailoring of information content and presentation (Winograd, 1996; Hackos and Redish, 1998).

Our notion of an "Adaptive WCSS" is much like a supportive collaborator who watches the user and foresees and provides whatever is needed while warning the user of pitfalls to be avoided. To interact successfully, human collaborators must establish several conditions. First of all, clear communication is only possible if there is a common vocabulary (shared semantics). Given that shared language, collaborators must agree on a model of the problem to be solved. It is this model of the problem that allows relevant information to be identified and communicated, alternative solutions to be identified or synthesized and evaluated, and the success of the process to be evaluated.

To achieve the vision of a WCSS, a collaborative relationship between human and computer system is required. The success of this collaboration will depend upon shared semantics and upon a shared model of the work to be done. This vocabulary should be that of the system's users, as espoused by the First-Person principle of WCSS (Eggleston & Whitaker, 2002). To be adaptive, a user-interface must be driven by evolving models of the work domain and of the human user of the system.

Our approach involves applying WCSS principles in a model-based adaptive user interface, essentially creating dynamic WCSS technology. User and work domain models must be able to predict what information will be most valuable to the user in the current context and determine how best to display this information. Additional information should be offered in the periphery of the display but with a cost of access inversely proportional to the inferred probability that access will actually occur. In other words, peripheral information more likely to be needed should be closer to the focus area of the display and therefore easier to perceive and bring into full focus (Card et al, 1999; Todd & Benbasat, 1999).

The underlying models tailor information content and display to work context as well as personal preferences, and perhaps other factors yet to be determined. Furthermore, the model of problem and human user maintained by the interface software must not be opaque. Rather, just as one person can ask a colleague how he arrived at a particular conclusion or why he decided particular information is relevant, a user must be able to ask the computer interface how a conclusion was reached, why a recommendation was made, or why a piece of information is included in the focus of the display.

Maintenance planning and logistics is a very dynamic domain in which many different problem-solving situations can arise, driving a corresponding amount of dynamic variability in information needs. We reduce the complexity of modeling information needs up front by allowing the user to adjust information layout and content in real time to suit the context in which work is being performed. Therefore, we only need to model context at a coarse grained level. The ACUITy architecture tracks changes made by users

as they customize the information design such that it is useful in specific problem solving activities. This capability establishes the environment in which automated tools can learn about, and extend the models to reflect, finer-grained work context. The following pages describe this architecture and its implementation.

2 Overview of the ACUITy Architecture

The ACUITy architecture has three major components: the Ontology, the ACUITy Controller and the User Interface Engine.

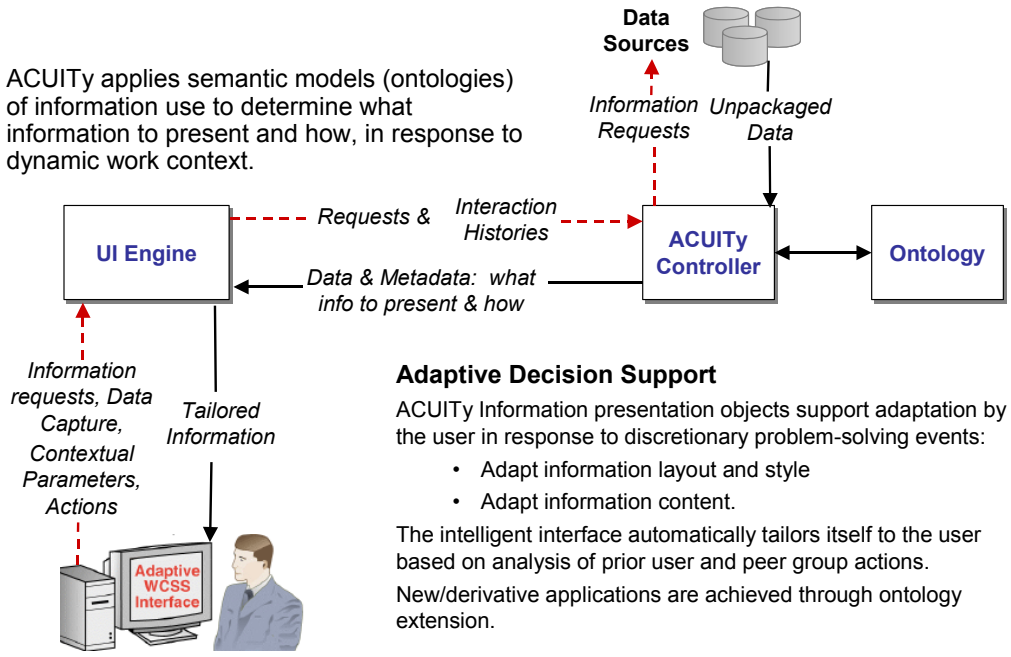


Figure 1: The ACUITy Architecture

2.1 Ontology as a Model Framework

UI design is always based on models of users and the work domain of interest. Designers make use of both Design Models, which can be implicitly understood by the designer or made explicit through some user-centered design process, and Implementation Models, which become part of the system itself, usually by hard-coding rules about what to display when. The WCSS framework essentially offers a way to develop more complete and work-centric models to support user interface design.

The models discussed above must be computational in order to guide automated or autonomic selection, bundling, summarization, display, etc. of information. Regardless of the particular computational approach chosen, a computational model can be broken down into the domain independent logic, which provides the formal structure and the rules of computation, and the ontology, which defines the kinds of things that exist in the

application domain and the domain-specific relationships between them (Sowa, 2000) (Figure 2). The ontology should capture information at more than one level. At the meta-level, the types of entities, relationships, and attributes of a domain of discourse are established. At a lower level, the instances of these types are established. Taken as a whole, the ontology provides the terminology (shared semantics) used by the system designers, implementers, and users.

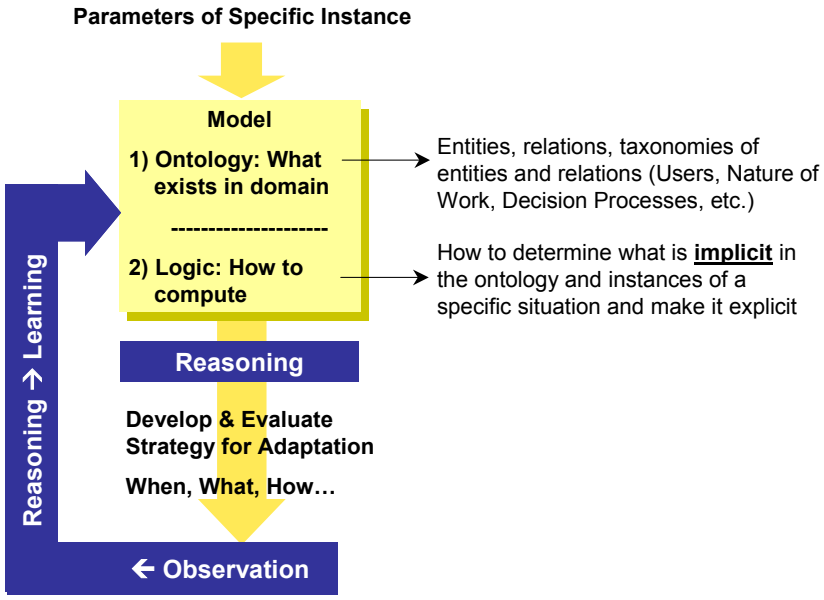


Figure 2: A learning, model-based system

It is natural for people to classify and otherwise organize things—it's how we reduce complexity. Classification, in conjunction with composition and other relationships, enables inheritance and provides tremendous parsimony in remembering and communicating. If I tell you that my daughter has a dog, you do not normally respond by asking me if it has hair, how many legs it has, etc. Rather I have only to tell you how this dog is different from dogs in general for you to have a pretty good understanding of my daughter's pet. More is communicated by the one word—"dog"—than by any discussion of this animal's uniqueness. This is only possible because we have mental models of the world that include similar prototypical dogs. The power of inheritance can be illustrated by another example. Suppose that I tell you that our exchange student from Africa has a civet as a pet and you respond that you have never heard of a civet. If I then tell you that a civet is a mammal about the size of a dog, you will be able to infer a large amount of correct information about the civet, even though you have only been told two simple facts.

It is precisely this organizational structure of knowledge that is captured by an ontology. Figure 3 shows how the inheritance of parts (dashed arrows on the right) can be driven by the classification relationships (on the left). Specialization (solid black arrows on the right) and pruning (red arrows on the right) allow for capture of additional complexities of a domain. The use of ontology, with inferential mechanisms such as inheritance, as a model framework is one approach to achieving extensibility of the work domain.

Informed by an ontological model of work in general and the particular domain in specific, new work elements might be added and classified. The relationships needed to model the new elements might be largely inherited, making the amount of customization needed to handle the new situation fairly small.

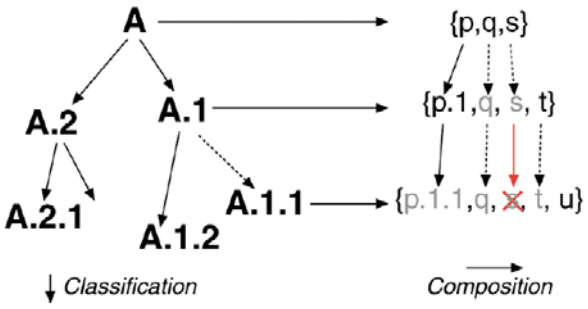


Figure 3: Inheritance, specialization and pruning of composition based on classification

The ACIUTy ontology is actually decomposed into a hierarchical set of related-sub models (Figure 4). There are several reasons for this. First, well-defined sub models may be reusable. This is a principle that is espoused by modular and object-oriented programming. Second, decomposition also allows sub models to be worked on in parallel by different modelers. A third reason is more fundamental. Humans have limited cognitive capacity, and to be understandable a model should not exceed that capacity. Decomposition into sub models allows the model builder or model user to focus on one manageable piece of the problem at a time. It is likely for this reason that work itself tends to be hierarchically decomposed by people, and by the systems that they build to assist with the work. The introduction of work-assistance artifacts can change the way the work is best organized and decomposed.

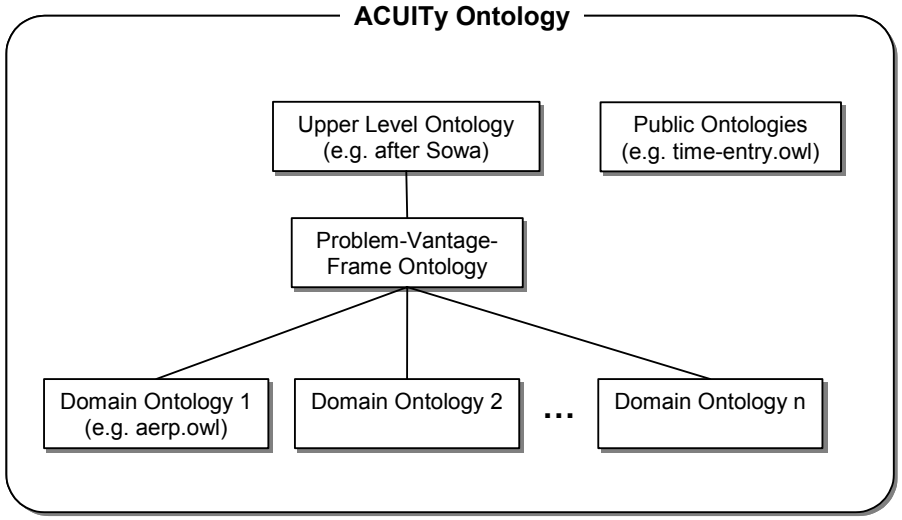


Figure 4: The ACUITy ontology hierarchy

In regard to this hierarchical organization, it should be noted that cognitive capacity appears to be extensible. In 1956, George Miller proposed that working memory has a

capacity of seven plus or minus two "chunks" (Newell, 1990). However, the content of a chunk can be small or large, depending on the degree of organization and accessibility of the information in the mind of the worker. This appears to be one of the primary differences between a novice and an expert in a given problem domain. Application of this principle in the model of users will be important if the information content and/or presentation is to be tailored to the skill level of the user. One challenge is that the "chunking" of the WCSS application must align with the chunking of the expert for optimal cognitive assistance to be realized and dissonance to be avoided. The creation of sub-models will conform to the needs of these various purposes in the design of a WCSS. Work will proceed in parallel on various sub models. Less experienced users may choose to work at a lower level in the work decomposition hierarchy than experts, matching the work model granularity to the size of their own cognitive chunks. Some sub-models may be reusable. In addition, hierarchical model structure is expected to be an important enabler for work and user model extensibility. Sub-models can be reconfigured, and adding additional sub-models can extend higher-level models. In this manner new ways of doing the same work or new work to be done can be included in the WCSS system in an evolutionary manner.

Presently, we have instantiated four levels of ontology in the ACUITy architecture:

1. An upper level ontology adapted from Sowa (2000)

2. The ACUITy "Problem-Vantage-Frame" ontology that proposes a high-level characterization of a Work-Centered Support System (WCSS) in terms of:

    a. The concept of problems and general problem types (more specific subtypes of which are defined in the domain-specific ontology)

    b. The concept of vantages into the information the user needs to solve the current problem (more specific subtypes of which are defined in the domain-specific ontology)

    c. How to organize the information into presentations that the user can manipulate

    d. Metadata about source data

    e. Mappings between data sources and presentations.

3. A WCSS specialization of the ontology that in effect "implements" the WCSS application for a particular work domain. This specialization defines the vantages, data sources and mappings that are specific to domain-specific work.

4. Publicly available ontologies (e.g. time-entry.owl) that we anticipate will be useful as we productionize the architecture.

Our primary focus in this report will be the ACUITy Problem-Vantage-Frame (APVF) ontology and its interactions within the architecture. Note that this ontology is still in a prototype state and, while we believe that we have achieved a level of stability in the design, some of the details are still being worked out. For example, we can currently

model legacy database interactions in terms of SQL queries and stored procedure calls but do not have a semantic model of database content that would enable automatic query construction based on a semantic description of the desired information.

Another area in which we have started to experiment but would like to do much more is in the area of ontology capture and maintenance. We are intrigued by Rector's (2003) approach to ontology development, which, among other things emphasizes the need for dividing ontologies between "refining" entities – entities, such as value types and partitioning concepts, that are fully dependent on other entities – and "self standing" entities. Rector's goal is to increase the ease with which consistent ontologies are developed and maintained through modularization and minimizing implicit information. We will not focus here on methodological issues with regards to ontology development, although we regard this as an important topic for future discussion. Rather, we will introduce concepts that are central to our general technical approach to adaptive decision support through ontology use.

2.2 Web Ontology Language (OWL) Concepts

The ontologies in the ACUITy architecture are expressed in the Web Ontology Language (OWL) (Dean and Schreiber, 2004), which is rapidly becoming the standard for semantic web applications. An OWL ontology fundamentally consists of four types of things:

1. Classes – abstract groupings of similar things, e.g., "mammal"
2. Individuals – references to the actual things themselves, e.g., "Lassie"
3. Property definitions (types) – abstract groupings of similar relationships, e.g., "spouse," and attributes, e.g., "age"
4. Property instances – statements, also called triples, consisting of a subject, predicate (property type), and object, e.g., "Joe has as spouse Susan", "Joe's age is 32".

Classes define the kinds of things that can exist in a domain of interest, while properties (that is property types) define the kinds of relationships and attributes that different classes of things can have. Instances of properties (statements) give additional information about the individuals. Properties can be of two types. DatatypeProperties describe a relationship between an individual and a data value, such as "Joe's age is 32," while ObjectProperties state a relationship of one individual to another, such as "Susan is the wife of Joe."

In principle, an ontology should formally define a class in an unambiguous way. In particular, subclasses of a given class should be defined in terms of differentia that allows individuals to be assigned to the appropriate subclass. OWL classes can be defined using property restrictions, and can give necessary or necessary and sufficient conditions. Classes may also be defined intentionally by listing the individuals that constitute the class, e.g., "Season" is one of "fall", "winter", "spring" or "summer."

OWL properties can have specified domains, which define the types of individuals that can have the property, and/or ranges, which define the types of individuals or data values that can be the objects of statements using the property. Unless the class is defined with a restriction on a property (e.g., it must have a particular value for a property, it must have

some values for the property from some class, or it must have cardinality requirements), then a property with the class in the domain is not required.

Different flavors of OWL have different levels of expressiveness: OWL Lite, OWL DL (Description Logics), and OWL Full. Greater expressiveness comes at the cost of greater difficulty in reasoning and less assurance of a computable result. For example, OWL DL does not allow classes or statements to also be used as individuals (be reified) so that additional information about them may be expressed.

While not required, OWL borrows the DL practice of expressing the classes and property types (terminology) separately from the individuals and statements about the individuals. The former is called the *tBox* while the latter is called the *aBox*. In ACUITy we store the aBox locally, augmenting it with new instance data after every session, as a form of learning. More about instance-based learning will be discussed in later sections.

The APVF concepts are captured in terms of OWL classes, individuals, property types, and statements. The APVF ontology establishes class hierarchies, allowing properties (restrictions and domain/range information) on a given class to be inherited from a superclass. The ontology also makes use of taxonomic ordering of properties into hierarchies to facilitate specialized reasoning over the APVF concepts.

Section 3 describes a subset of the OWL classes, properties and restrictions that we defined in the APVF ontology. We outline classes of prime significance in defining an adaptive work-centered application, their roles and interactions in the APVF framework and specific properties that are pertinent to those classes (note that this list of properties, some of which may be interpreted from superclasses, may not be exhaustive).

2.3 The ACUITy Controller

The ACUITy Controller is a Java class (along with supporting classes) that provides an API to the ACUITy Problem-Vantage-Frame (APVF) ontology knowledge base described above. The ACUITy controller queries the ontology to understand where to find data, how to obtain it, and how to bundle it. The controller also accepts inputs from clients and updates the ontology accordingly. While the ACUITy controller is largely domain-independent, we have designed it to be sub-classed if needed to create domain specific APIs.

The ACUITy Controller currently performs four main functions. First, it provides a tell/ask interface to the underlying ontology, which includes classes, properties and instances (Figure 5).

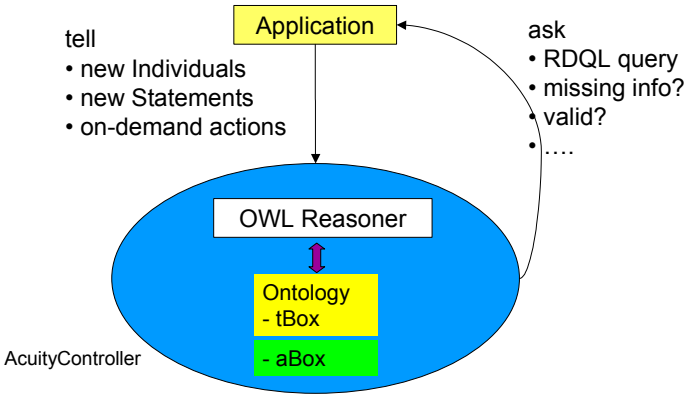


Figure 5: Tell/Ask features in the ACUITy Controller

Second, the ACUITy Controller selectively persists instance data, enabling memory and simple learning (Figure 6).

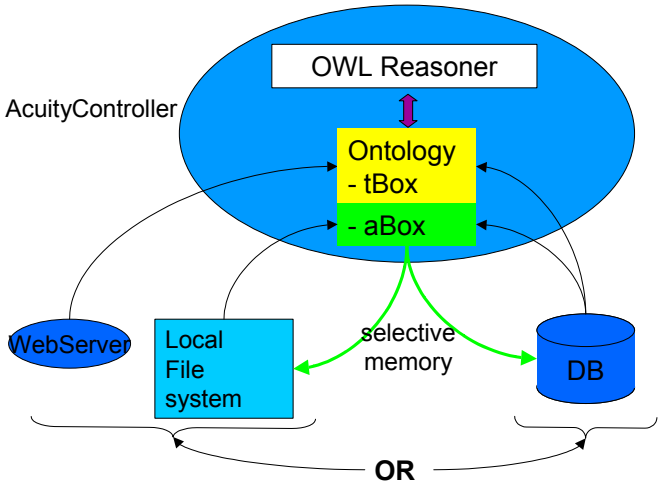


Figure 6: Selective persistence in the ACUITy Controller

Third, the ACUITy Controller encapsulates and extends reasoning capability. The Jena library (Jena) provides encapsulated reasoning; the Jena transitive reasoner is the default OWL reasoner (Figure 7). An example of special purpose reasoning provided by extension in the ACUITy Controller is the transparent pass-through of legacy data (Figure 7).

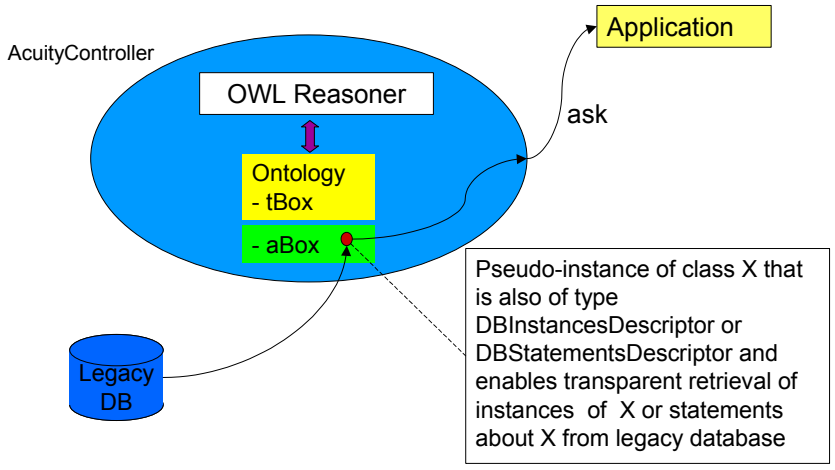


Figure 7: Transparent pass-through of legacy data in the ACUITy Controller

Finally, the ACUITy Controller integrates procedural knowledge (actions) by association with certain OWL classes and properties. OWL is a declarative language: you can declare what action to take, but OWL itself does not execute the action. Some examples of the type of procedural actions performed by the ACUITy Controller include:

- User actions: e.g. create a new session ("frame").
- Auto-populating a new frame with vantages and vantages with presentation objects.
- Mapping functions for graph presentation. For example, transforming a query into a data series (e.g. create a candle chart from a legacy query), or transforming raw data into presentation attributes (e.g. numeric values to graph labels).

One can extend the procedural actions provided by the ACUITy Controller by creating user-defined scripts, which include new instances of action classes in the shared semantic model. Scripts are discussed in more detail in Section 3.15.

2.4 User Interface Engine

The role of the User Interface Engine is to accept the ontological information obtained from the ACUITy Controller and generate the client application interface. It sits between the end client application and the ACUITy Controller (or application specific subclass of the same) and is meant to be the single interface point between the two entities.

The UI Engine has three main responsibilities (Figure 8):

1. It is the interface between an application and the ACUITy Controller. As such it interacts with the controller to request information from the ontology and update information in the ontology on behalf of the client application. It masks the ACUITy Controller API so that client applications do not have to know how to interact with the controller, or have the domain knowledge of how to interpret, marshal, and transform ontology information.

2. It generates user interface display objects to represent the ontological information retrieved from the Controller. These model objects are a client platform agnostic representation of the user interface, which allow the UI engine to render many different types of client interfaces.

3. It renders the UI model objects to the specific client application platform. For example, a web client that uses the ACUITy architecture would receive a well-formed HTML document from the UI Engine. Another application (such as an Eclipse client) would use the same model objects and return Eclipse objects for the interface. The set of client-specific renderers allows the UI Engine to generate the user interface for the client application, so that minimal application development is needed to construct the client.

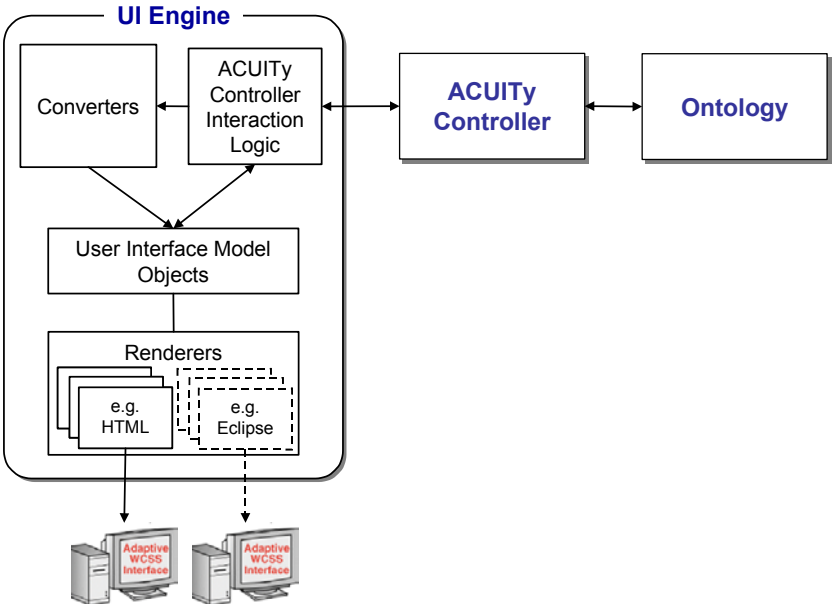


Figure 8: The User Interface Engine

The UI Engine interacts with the ACUITy Controller in the following manner. First, using the information provided from the client application (the request object) the UI Engine sends update information to the ACUITy Controller. The current UI model objects are also updated to reflect the same changes. At this point the flow of execution branches, depending on the updates that were made.

If the updates made do not alter the content (but could alter the presentation) of the current information set, then the current UI model objects are also updated to reflect the same changes. Such changes include repositioning and/or changing color, device type, etc. The model objects are then passed to the client-specific platform renderers to create the interface for the client, which is then passed back to the client application.

If the updates alter the information content, then a new set of information is acquired from the ontology via the ACUITy Controller. Such updates include (but are not limited to) changing the focus vantage, changing a property value and/or supplying missing

information. The new information content is converted into the UI model objects and then passed to the client-specific platform renderers to create the interface for the client. This interface is then passed back to the client application.

2.5 An Introduction to the APVF Ontology

Eggleston and Whitaker (2002) propose the Problem-Vantage-Frame principle as a general guide in designing work-centered support systems, as follows:

> "The work-centered approach treats work as an unfolding series of situated problem-solving events. Effective interfaces must be attuned to these events so as to both portray the situation and work problems, while also providing affordances in terms of the work itself (e.g. key referents, actional objects). We do this by developing for each intrinsic problem/event category a specification for the information subspace with which users can adequately manipulate the referential coordinates, level of detail, and level of abstraction for work domain variables. The goal is to have the interface accommodate the vantage point a user may adopt to meet the current situations. A WCSS frame instantiates such a vantage with specific display and control elements. A single WCSS can include multiple such frames (Eggleston, et al., 2000). Because this design strategy progresses from problem to vantage to frame, we refer to it as the *Problem-Vantage-Frame* principle."

We believe that if we can explicitly model the Problem-Vantage-Frame (PVF) concepts as a central controlling entity in the ACUITy architecture, we can better enable adaptive responses to changes in work context and easily extend user interface designs into new problem domains. This goal requires us to formally represent the PVF concepts as they relate to pragmatic issues pertinent to UI design.

We do this in the ACUITy Problem Vantage Frame (APVF) ontology by formalizing the concepts Frame, Vantage, Presentation Object, Work Domain Information Object, DataMap, Filter, Script, and Property and Parameter (Figure 9). In this section, we introduce these general concepts. In Section 3 we describe each of the concepts in more detail.

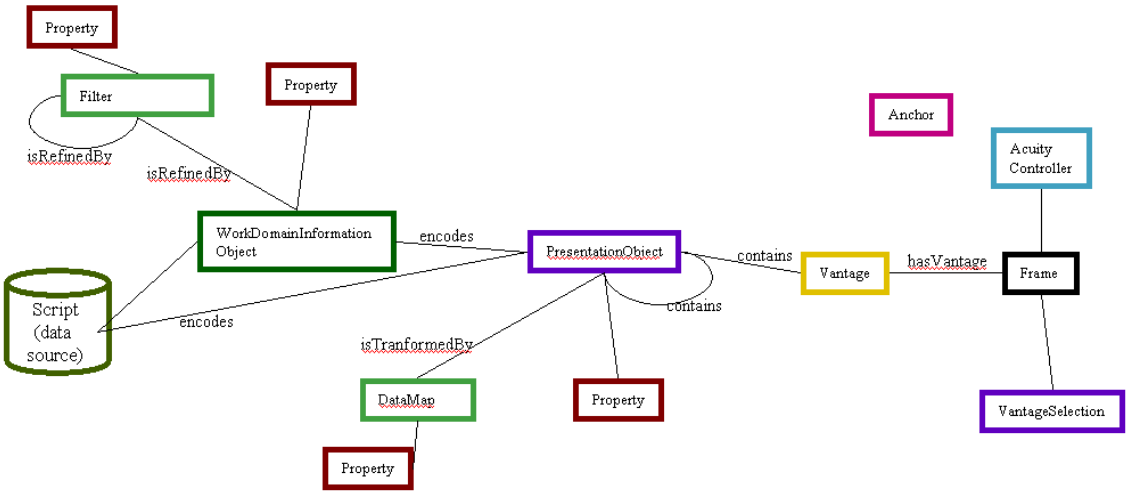


Figure 9: Key concepts in the Problem-Vantage-Frame ontology

Problem

Through extensive work analysis, we characterize the work being done in a domain of interest in terms of the problems the user is working on and how, within the domain, problems evolve as they are being solved. We currently define a "problem" as something perceived not to be as desired; i.e., a state of affairs that requires the user to take action to change some aspect of the work domain. At this point in our technology development, we have not encountered a need to explicitly represent domain-specific problem types in our ontology. The problems we have encountered in our application work domains thus far have had a one-to-one alignment with vantages (defined below). This is probably due to the nature of the domains we have modeled and we believe we will soon encounter the need to represent problem types at lower levels of detail. Note that this will require some way of observing work domain context and user actions to infer problem state, an issue that will benefit from future research.

Vantage

A Vantages is a "window" into work domain information that provides the user with the particular perspective needed to solve a problem or problem set. More formally, a vantage is a collection of presentations of information that are relevant for a particular problem, along with their level of relevance, potentially associated with a preferred position and size in the layout of a display.

Frame

A WCSS frame "instantiates a vantage with specific display and control elements" (Eggleston and Whitaker, 2002). We interpret this as the properties of a session with one or more Vantages, each of which will have an associated problem set and a set of relevant information objects. In other words, a Frame brings a set of Vantages into relation via a set of common properties to mediate a particular work session. For each domain, the frame is specified as 'containing' vantage objects. The ACUITy Controller manages user sessions in terms of Frames. With each user session, the user is given the option of

creating a new session (Frame instance) or picking up their work from an existing session (Frame instance).

We enable a single frame to be populated by multiple vantages, one of which occupies the focus of the display while other secondary vantages are easily accessible in the periphery. We do this because a user may be working on a problem set that requires somewhat different and discretionary insight into the problem domain without losing the overall frame of reference.

The following concepts are extensions of the Eggleston, et. al. PVF framework.

Presentation Object

In the APVF ontology, vantages contain presentation objects that are displayed to the user. A presentation object is a description of how to present information to the user. Presentation objects have a "presentation nature", the value of which is specified as a particular type of display object. Though the type of the display object sets the base presentation style, the user can be allowed to modify its look and feel. User choices can be preserved in the ontology for later reasoning and learning about how individual users and user types prefer to view data in various circumstances. This addresses the "First Person" principle of work-centered design.

Work Domain Information Object

Instances of presentation objects "encode" information for presentation to the user. This information is normally conceived of as a work domain information object. The exact instructions for retrieving information, whether from the ontology or from an external data repository, is captured in a script. If desired, the domain information objects may be thought of as directly encoding a script. Work domain information objects may also be refined by filters, which mask some of the information in the data source. Filters may be chained together in order to obtain the desired information content. From the point of view of the UI Engine, classes of things that are defined in legacy databases or other external data sources can be made to appear as part of the ontology without having to re-define and re-classify them. Work domain information objects are typically domain-specific representations of data retrieval mechanisms such as URLs, database queries, stored procedures, file reads, etc.

DataMap

Data maps define how to transform source data or work domain information object to the format required by a presentation object. A set of standard maps is provided for the most common transformations. A map also provides for the specification of custom plug-in transformations.

Script

A script is a set of instructions. Scripts include queries and of dates to the ontology, queries, stored procedure calls, and updates to relational databases, and methods written in Java. In general, scripts either specify the retrieval of information to be encoded in an information object (as shown in Figure 9) or they specify a side effect.

Properties and Presentation Parameters

In general, we use the term "property" here to mean the value of an OWL property associated with objects in the ontology. Properties represent values that are used by the ACUITy Controller and UI Engine to obtain, compose and render information. Some of the properties are used only by the ACUITy Controller (e.g., database connection information), some by the Controller and UI Engine (e.g., whether a display object is visible), and others are used by the user (e.g. window position and size). We refer to properties that control the visible layout of information as "presentation parameters". User-modified values are passed back to the ACUITy Controller and recorded in the ontology for deriving information display properties for future sessions.

The user may be able to manipulate some parameters (e.g. position, size, etc.) while other parameters have their value restricted within the ontology and cannot be modified by the user. As an example of the latter, the color of a data series in a graph may be given a default value that can be modified by the user and ultimately learned from the user, or it may be restricted to a particular color, e.g., red, in which case the user sees the series as red but cannot modify it.

2.6 Adaptation in the ACUITy Architecture

Users Finish the Design

The ACUITy architecture enables two types of adaptation. First, it enables adaptation by the user in response to the particular context of work by enabling the user to modify their view on the work domain as they devise new problem-solving strategies. This type of direct adaptation is generally regarded as a critical need in open-ended, information intensive work (Vicente, 2000), but something that is typically difficult to support well in static information design. Thus, we see users in many open-ended problem-solving domains using applications like Excel and Access, which allow for very flexible composition, organization and exploration of datasets. These applications lack many of the benefits of a web-based environment, but in order to reap the benefits offered by a web-based environment we must be able to address the need for users to modify data visualizations according to the specific context of their work at hand – context that we undoubtedly cannot fully anticipate up-front in design.

We support user adaptation by allowing users to manipulate presentation parameters and select and deselect presentation objects associated with vantages. In other words, they can decide what information they require to solve a particular problem – defining the vantage they need on the problem domain – and they can also change the characteristics of the information display in order for them to interact with data more effectively.

In the future, we would also like to support users in performing ad hoc data queries. We anticipate that new sources of data will become relevant to users during the course of their work. We would like users to be able to map to these sources of data without architecture or UI re-design. Another subject for future development is a custom

presentation object that allows the user and/or application modeler to implement custom Java classes to render custom presentation objects.

Learned Defaults

Second, the ACUITy architecture enables automated adaptation through learning. One can envision two types of ontology-based learning. The most easily applied, and the one currently implemented in ACUITy, is instance-based learning. As a domain application is used, instance data accumulates. From this instance data, one can see how particular users or groups of users chose to interact with the application. Thus as users customize vantages and presentation objects, we seek to use the ontology as a memory that allows new users and users in new work contexts to take advantage of what has been done in the past either by the current user or by wider peer groups. This past history creates the opportunity for a reasoner to infer what presentations are most appropriate based on new information that was unavailable during the initial design of the user interface.

A second type of learning, which is not yet implemented in ACUITy, involves automatically or collaboratively extending the ontology to include new classes and/or property types. This will cause new metadata to be available that was not available in past sessions. The new metadata will allow either finer-grained distinctions in the model or the extension of the model in response to evolution of the work domain. For example, identifying a set of instances that have something in common as a new class would allow defaults and learned values to be applied to a more narrowly defined set of future instances.

2.7 Putting the general concepts together

A domain-specific application is characterized by an ontology that has been specialized in terms of:

- A specialization of the frame class for the domain,
- Vantages that may be presented to the user,
- Presentation objects that are to be rendered within each vantage,
- Sources of data for each display object, possibly including filters,
- Maps, if any, defining how to transform the data from its source format to a format compatible with the display object,
- Input constraints, e.g., possible answers, on each interaction object, and
- The default values used to establish a starting point while supporting adaptability and learning.

From the frame, the ACUITy Controller identifies the current vantage and from there the corresponding presentation and display objects. For each display object, the controller follows the ontology definition back to the data sources. Data from the data source is "mapped" to generate the data displayed in the display object. The display objects, the

appropriate properties and the data series are bundled together and passed to the UI Engine, which then renders the information for the user. The UI may have user interaction points via interaction objects. The UI Engine will collect the interaction values from the client and pass them to the ACUITy Controller, which in turn updates values in the ontology. When the user logs out of the application, the ACUITy Controller automatically saves the session data. At the beginning of each session the user is prompted to select a saved session or create a new one. These saved values are also available for auxiliary reasoning to perform such adaptation as automatically modifying the display based on users' preferences.

3 The ACUITy PVF Ontology Components

The following sections describe the above concepts in more detail and documents how we have implemented them in the owl APVF ontology. The development version of the APVF ontology is included in Appendix C for reference.

3.1 AcuityController

"ACUITy Controller" has two meanings in this paper. In one sense we have a set of Java code that is our ACUITy Controller (implementing the runtime management of the APVF ontology and interfacing with the UI Engine) and in the second sense it is the OWL class, AcuityController, an instance of which represents a session. We use the latter here, to describe the concept of a session having a single user and frame. All other references to ACUITy Controller in this paper refer to the Java class that implements the runtime management of the ontology and interfacing with the UI Engine.

The AcuityController as a concept (owl:Class) is found in the APVF ontology and each client session will instantiate an instance of this class. It provides the root for persistent memory of past sessions. Each instance of AcuityController is characterized by (has the properties of) a specific frame and user.

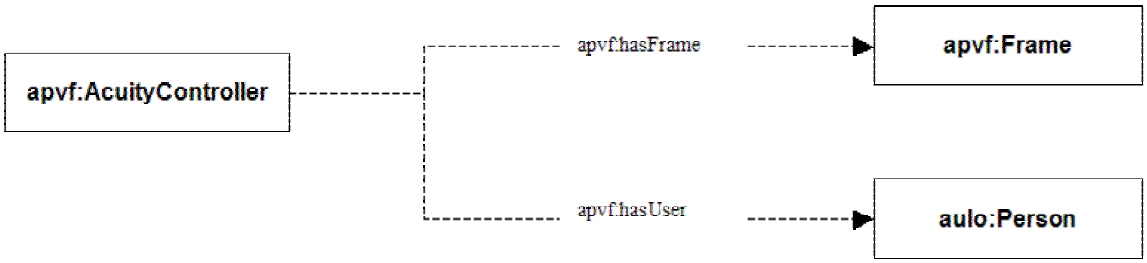


Figure 10: ACUITy Controller as a session

AcuityController Properties:

- hasFrame – the frame for the current session (instance of AcutiyController)
- hasUser – the User for the current session (instance of AcutiyController)

AcuityController Restrictions:

- The controller has a single frame
- The controller has a single user.

3.2 PresentationObject

A PresentationObject represents something that is presented to the user. Presentation objects in an application are defined as domain-specific subclasses of this class, such as graphs, tables, etc. Common to all presentation objects are the following properties:

PresentationObject Properties:

- hasPosition – the position and size for the rendered object
- isDisplayed – a Boolean indicating if the object is rendered viewable within the client user interface. It will have different affects on the presentation depending on what subclass it is being instantiated for.
- containsPresentationObject – The specialized subclasses of PresentationObject use this property to characterize each subclass in terms of how they nest presentation objects.
- hasPresentationNature – this property defines the general presentation style for the object. The presentation nature of an object triggers the appropriate handling in the UI engine.

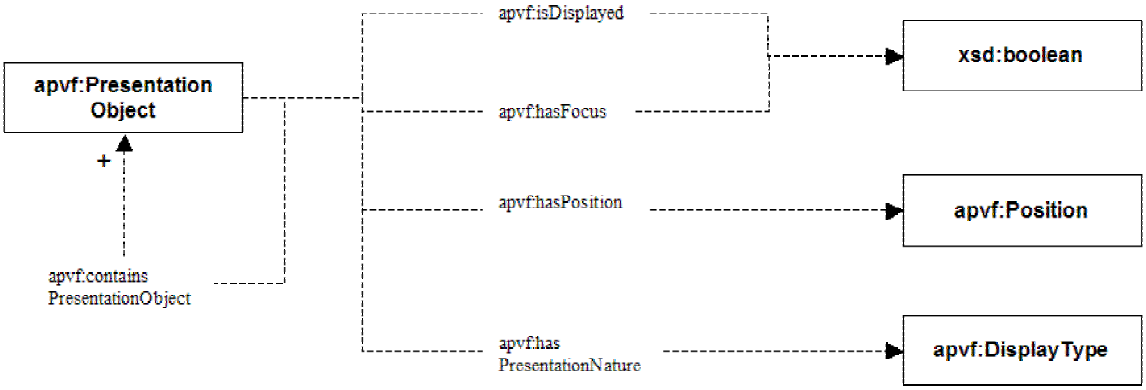


Figure 11: PresentationObject Properties

Of key interest is the property hasPresentationNature with range DisplayType. Display types have the subclasses and instances shown in the table below. Each of instance of DisplayType implies a certain style of rendering and user interaction.

Subclass of DisplayType	Instances (Individuals)

Graph	Graph2D
	Graph3D
GroupDisplay	InteractiveGroupDisplay
	NonInteractiveGroupDisplay
Series	area
	horizontalBar
	line
	pie
	point
	pointAndArea
	pointAndLine
	VerticalBar
InteractiveTabularSign	CheckBox
	DropdownListBox
	ListBox
	Radio
	TabsList
NonInteractiveTabularSign	ScrollingTable
InteractiveText	Hidden
	TextArea
	TextBox
NonInteractiveText	DisplayedLink
	DisplayedText

Subclasses of PresentationObject:

- Frame (see Section 3.4)

- Vantage (see Section 3.4)
- DisplayObject (see Section 3.5)
- InteractionObject (see Section 3.13).

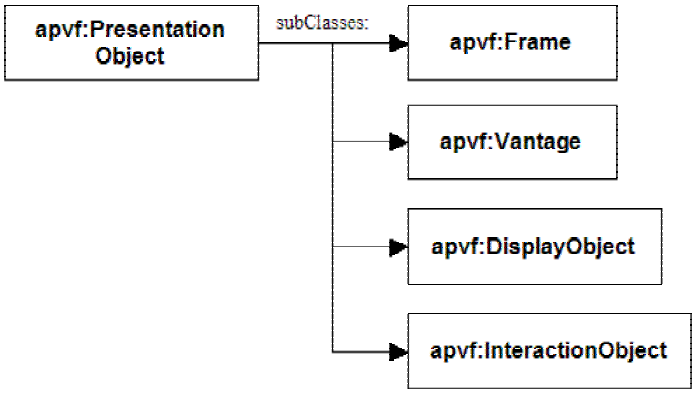


Figure 12: PresentationObject subclasses

3.3 Presentation Parameters

When a class is characterized by a presentation parameter, we specify a hasPresentationParameter property. Currently there are five types of hasPresentationParameter properties that correspond to the value types: Boolean, floating point number, integer number, string, and object.

- hasPresentationParameterBoolean: e.g. allowsMultipleSelection; displayGridlines; hasFocus, etc.
- hasPresentationParameterFloat: e.g. opacity; hiThreshold; loThreshold, etc.
- hasPresentationParamterInteger: e.g. xPos; yPos; zPos; width; depth; height; blueVal; greenVal; redVal, etc.
- hasPresentationParameterObject: e.g. hasPresentationNature; hasColor; hasPosition; hasSlice; hasColumn; hasHighlightRegion,etc.
- hasPresentationParameterString: e.g. title; yAxisTitle; xAxisTitle; linkDisplayText; zAxisTitle; appliesTo, etc.

While most presentation parameters are single-valued, composite parameters contain at least one and usually more sub-parameters. In other words, a composite parameter is a parameter whose object is another parameter (e.g. position has x and y positions as objects).

There are two reasons for composite parameters. The first reason is for convenience of organization in the ontology (for example to aggregate color from red/green/blue values). The second reason is to allow for sharing of parameters among presentation objects, such as can be useful in “brushing” data on graphs (see Section A8.3). The value of an OWL

DataTypeProperty, e.g. a number or a string, is not a uniquely identified element of an OWL ontology and therefore cannot be shared between presentation objects. Rather, the value would be saved with each instance of the modified presentation object and there would be no way to represent (and remember) that the two presentation objects have the value in common. A good example of this is the class Color, the range of the hasColor property. It is quite conceivable that in some kind of brushing operation (highlighting the same data in multiple views), two presentation objects might want to use the same color. The Parameter class Color is a composite parameter with the datatype properties alpha, hasBlueVal, hasGreenVal, and hasRedVal. If the datatype properties of a particular Color (e.g. Aqua) are changed, then they are changed for all individual presentation objects characterized by that particular color.

The current subclasses of Parameter are shown in Figure 13. Throughout Section 3 we will describe various members of the Parameter class in more detail.

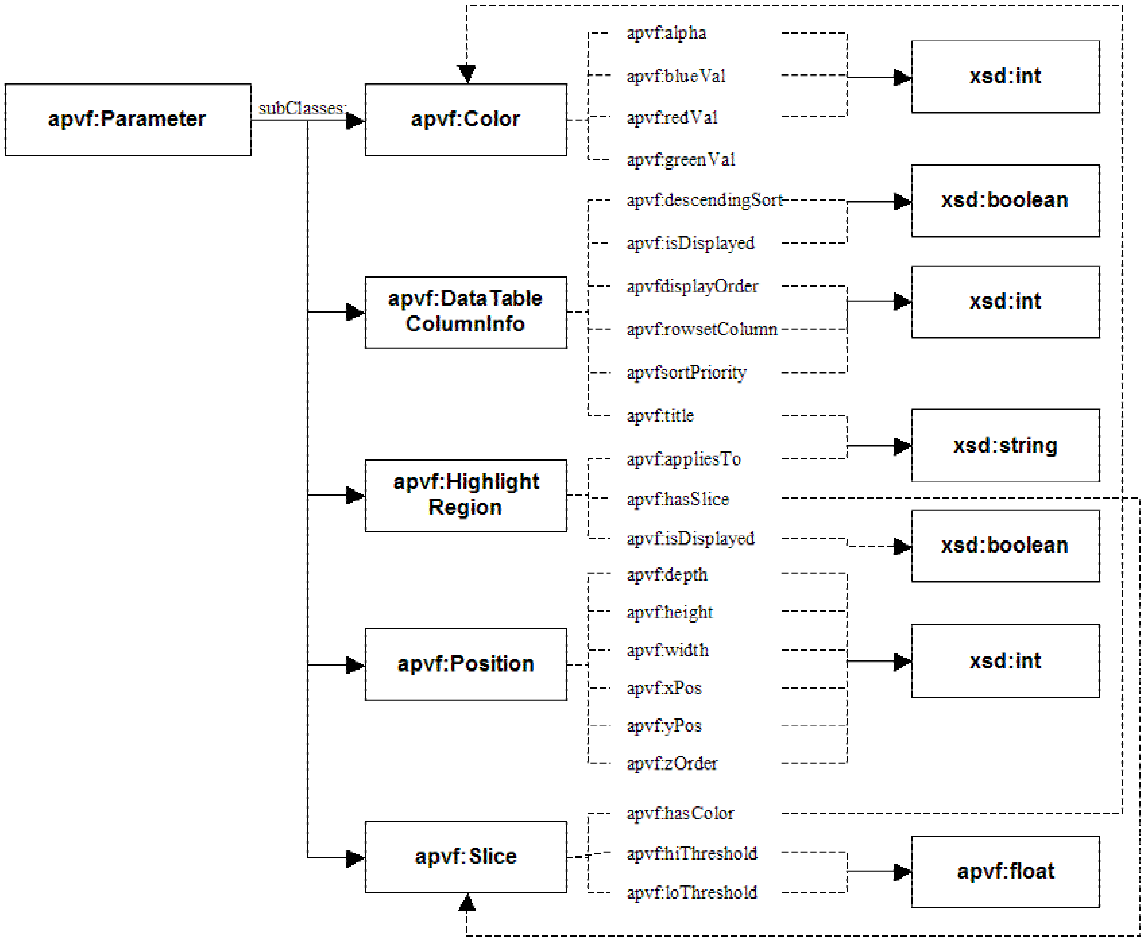


Figure 13: Parameter subclasses

3.4 Vantage and Frame

A Vantage is a "window" into work domain information that provides users with a particular perspective needed to solve a problem or problem set. A Frame brings a set of vantages into relation via a set of common properties to mediate a work session.

Vantage Properties:

The general Vantage class does not have any properties of its own. An application domain will normally subclass Vantage and add properties and restrictions appropriate for that application.

(Inherited Properties:)

- containsPresentationObject – this property characterizes a vantage in terms of the presentation objects that make up the vantage. Typically, each presentation object will contain information that is to be rendered to the user.

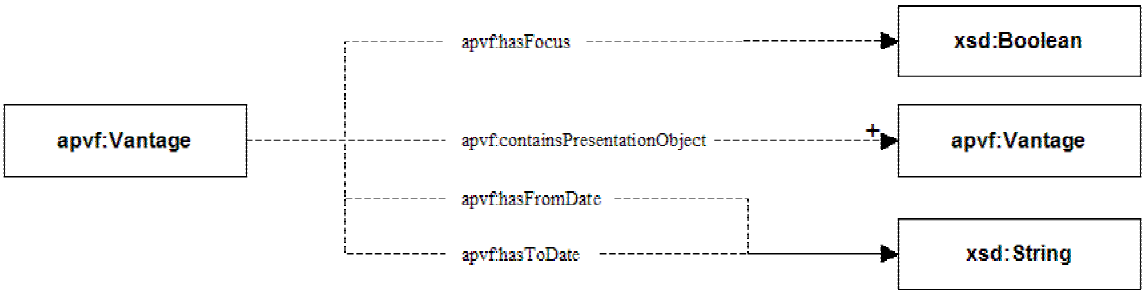


Figure 14: Vantage properties

Frame Properties:

- hasAskUserRDQLValueFilter – an RDQL query string that selects the existing Frame instances available to the current user (see section 3.14 for an overview of RDQL)
- hasFocusVantage – this property characterizes the frame in terms of the vantage that is currently being presented to the user.
- hasVantage – this property characterizes the frame in terms of all the vantages that are candidates for being presented to the user.

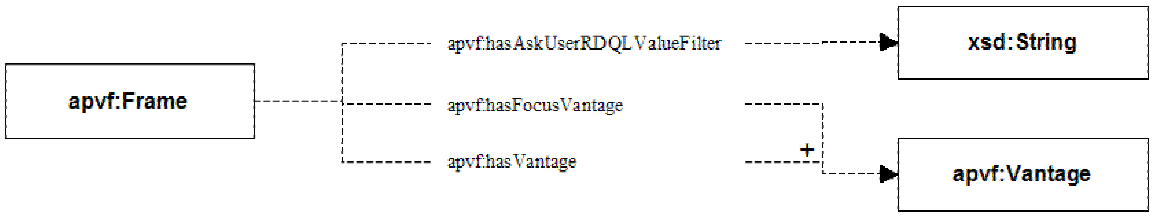


Figure 15: Frame properties

Frame Restrictions (Necessary Conditions):

- hasFocusVantage has exactly one value – this single focus vantage is the vantage that will be rendered.

- At least one of the containsPresentationObject properties is of type VantageSelectionObject – the VantageSelectionObject is described in detail below. In essence this restriction means that a selection mechanism needs to be presented to prompt the user to select a vantage.

The APVF ontology does not define subclasses of Frame or Vantage. For most applications, there will be domain specific extensions of the APVF ontology that will subclass the Frame and Vantage classes to specify domain-relevant properties.

3.5 DisplayObjects

Beyond Frame and Vantage, we currently identify two other subclasses of PresentationObjects: DisplayObjects and InteractionObjects. Display Objects represent ways to render information for the user, whereas InteractionObjects represent ways to collect (accept) information from the user.

DisplayObject Properties:

- hasHighlightRegion – a highlight region defines an area of the graph that is rendered with a highlight or 'brushing' effect. Shared highlight regions provide coordination between various presentation objects' highlight regions. If a user repositions a highlight region in one presentation object, any shared highlight regions in other presentation objects will be updated appropriately (see Section 3.11 for a more detailed description of highlight regions).

Subclasses of DisplayObjects

- GraphObject – defines a graph-styled rendering of the information

- DataTable – defines a table-styled rendering of the information

- DocumentObject - defines a formatted text-styled rendering of the information

- DataSeries – defines the discrete sets of data that will be used in a GraphObject

- SimpleValueObject – defines a simple name/value pair (subclasses have values for each of the xsd datatypes)

- DisplayGroup – defines a grouping of display objects that are positioned relative to the DisplayGroup and move together.

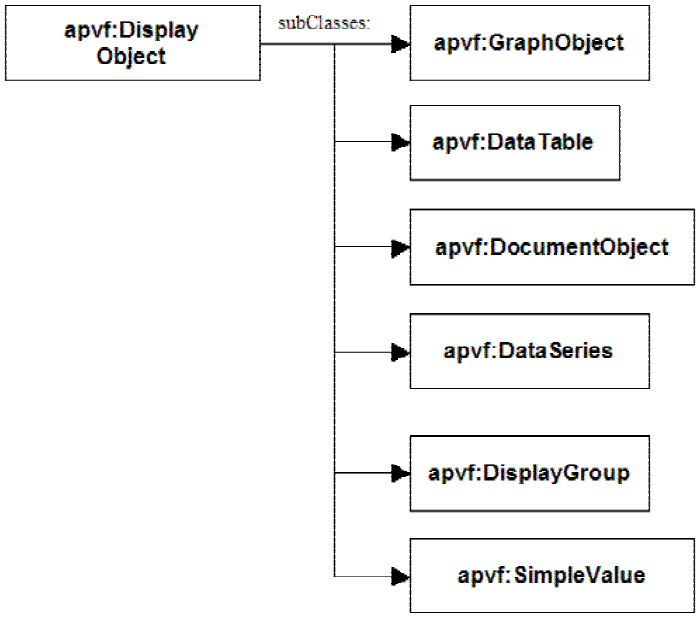


Figure 16: DisplayObject subclasses

3.6 Graph

Graph is a subclass of DisplayObject. We envision graphs to be either a 2D or 3D rendering of data series. We focus here on 2D graphs; we expect that 3D graphs will simply be extensions of 2D graphs.

A graph contains one or more data series. A data series is a set of data points that are rendered in the graph with a common set of characteristics. With each graph definition, there is a graph-level set of properties that describe rendering characteristics that are shared across the data series; and for each data series there is a data series-level set of properties that describe rendering characteristics specific to the data series.

Graph Properties:

- isDisplayed – at this level, this flag indicates whether the graph is to be rendered visible or appear as a minimized icon.
- displayXgridlines – a Boolean indicating whether or not to render x axis gridlines in the graph.
- displayYgridlines – a Boolean indicating whether or not to render y axis gridlines in the graph.
- displayZgridlines – a Boolean indicating whether or not to render z axis gridlines in the graph.
- xAxisTitle – the title to display for the x Axis.
- yAxisTitle – the title to display for the y Axis.

- zAxisTitle – the title to display for the z Axis.
- title – the title for the graph.

(Inherited Properties:)

- containsPresentationObject – the mechanism that is used to designate a graph as containing one or more dataseries. Each specialized subclass of the Graph object (specialized to define a domain specific graph), will be restricted to contain one data series presentation object for each data series that is to appear in the graph.
- hasPresentationNature – the value of this property is restricted to either graph2D or graph3D. At this point in time we have not implemented a renderer for 3D graphs.
- hasHighlightRegion – see Section 3.11 below.

Figure 17: GraphObject properties

Graph Restrictions (Necessary Conditions):

- At least one of the values of the containsPresentationObject property is a DataSeries.

DataSeries Properties:

- displayOrder – the relative order across data series for rendering each data series.
- hasColor – the color to use for the data series.
- title – the title for the data series.

(Inherited Properties:)

- hasPresentationNature – indicates whether to display the series as point, point and line, line, area, point and area, horizontal bar, vertical bar, or a pie chart.
- isDisplayed – at this level this flag indicates whether the series should be rendered in the graph.

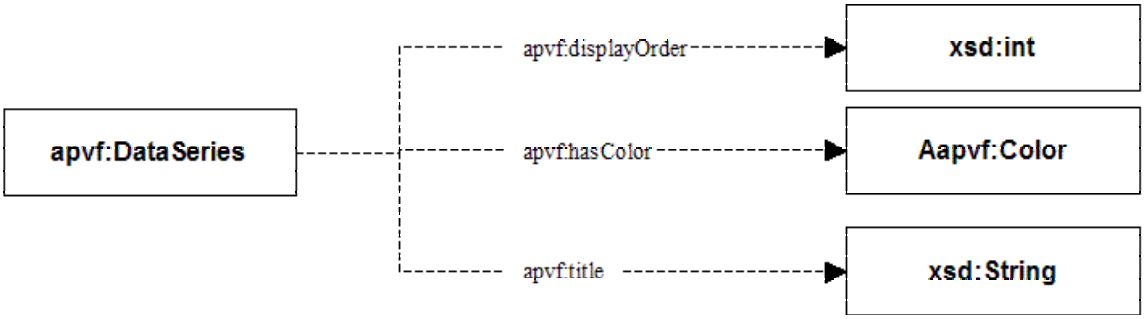


Figure 18: DataSeries properties

DataSeries Restrictions (Necessary Conditions):

- a DataSeries contains a single hasPresentationNature property and it is of type Series (see Section 3.2).
- at least one of the values of hasColor property is of type Color (see Color definition below).
- displayOrder has exactly one value.

Subclasses of DataSeries:

- *SimpleDataSeries*: The data source for a SimpleDataSeries is made up of a list of (x,y) value pairs. For each (x,y) pair in a simple data series, the ACUITy Controller uses the first value for the x value and the second value for the y value.

  SimpleDataSeries Properties:

  - isGeneratedBy – specifies the stored procedure or SQL statement that provides a set of values for plotting.[1]

  SimpleDataSeries Restrictions (Necessary Conditions):

- the value of the isGeneratedBy property is either a StoredProcedure or SQL Statement[2]

- *MappedDataSeries*: MappedDataSeries has an additional isGeneratedBy property that identifies a Data Series Map, which in turn, defines how to transform the source data to a set of (xmin, xmax, ymin, ymax) values for plotting. Please refer to Section 3.10 for a description of how these maps are handled.

  MappedDataSeries Properties:

- isGeneratedBy – specifies the Data Series Map that generates the set of values for plotting.

  MappedDataSeries Restrictions (Necessary Conditions):

[1] This could be extended to a script.

[2] This could be extended to a script.

- The value of the isGeneratedBy property is a DataSeriesMap

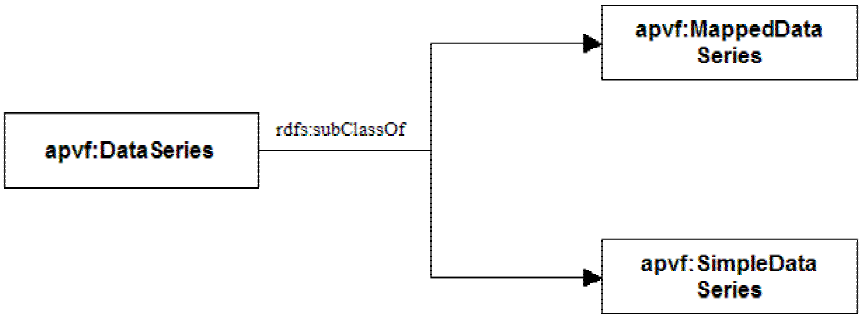


Figure 19: DataSeries subclasses

3.7 DataTable

DataTable is a subclass of PresentationObject that renders a two-dimensional array of data. The presentation and adaptable characteristics of the tables are either defined as table level properties or column level properties. The column level properties apply to a specific column in the table.

DataTable properties:

- alternateRowShading – when this is true, the table will be displayed with alternate rows shaded
- topVisibleRow – the top row to be displayed in the table, which identifies scroll position.
- hasColumn – Each column in the table allows a certain level of user and/or domain specific formatting: turning on/off the display of the column, sorting, naming, etc. The DataTableColumnInfo class manages the rendering of each column. This property specifies the DataTableColumnInfo objects for the table. When the domain specific ontology does not explicitly define the DataTableColumnInfo objects for a column, the runtime environment automatically creates them. See DataTableColumnInfo below for more details.
- title – the title for the table

(Inherited Properties):

- hasPresentationNature – establishes the presentation nature of the object as a table.
- hasHighlight Region – see Section 3.14.
- isDisplayed – at this level, this flag indicates whether the table is to be fully rendered or appear as a minimized icon.

Figure 20: DataTable properties

DataTable Restrictions

- hasPresentationNature is set to scrollingTable.

The *DataTableColumnInfo* class defines how to render each column in a table. The domain-specific application can explicitly identify the DataTableColumnInfo class to be used for a particular column. If no DataTableColumnInfo is defined for a column, the ACUITy Controller will automatically create one.

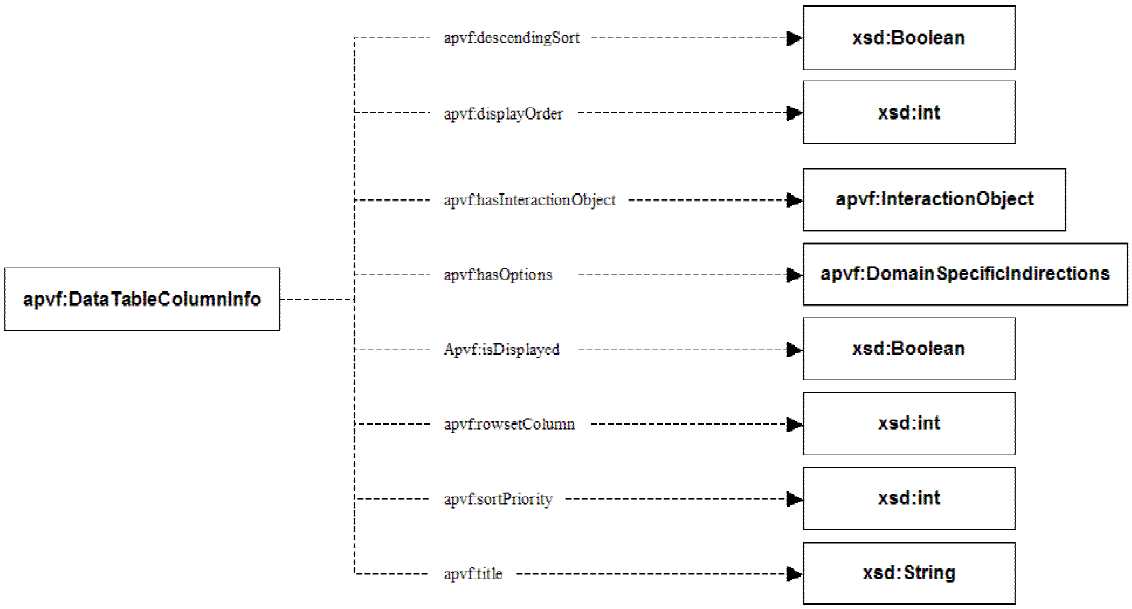


Figure 21: DataTableColumnInfo properties

DataTableColumnInfo properties:

- descendingSort – indicates the use of a descending order if the column is to be sorted.
- displayOrder – indicates the order for rendering the columns.
- hasInteractionObject – each cell in the column can be 'active' in terms of supporting interaction with the user. This is specified via this hasInteractionObject property.

When a column has an InteractionObject, that object defines the interaction. See section on InteractionObject below.

- hasOptions – identifies any TableCellDecoratorOption(s) that can be applied to the column values. See the section on TableCellDecoratorOptions for more information.
- isDisplayed – indicates if this column is hidden or displayed.
- rowset column – defines the column from the original or mapped data set that the controller is to use to find the column values.
- sortPriority – if non-zero, this indicates the nesting order for sorting multiple columns.
- Title – the title for the column.

Subclasses of DataTable:

- *SimpleDataTables* expect that the data source is simply a two-dimensional array of values that are rendered directly to the user.

  SimpleDataTable Properties:

  - isGeneratedBy – specifies the StoredProcedure or SQL statement[3] that provides the set of values for rendering.

  SimpleDataTable Restrictions (Necessary Conditions):

  - The value of the isGeneratedBy property is either a StoredProcedure or SQL statement.[4]
- *MappedDataTables* define how to transform the source data to a two-dimensional array. They have an additional isGeneratedBy property that identifies a DataTableMap that, in turn, defines how to transform the source data to 2D array of values for rendering. Please refer to Section 3.10 for a description of how these maps are handled.

  MappedDataTables properties:

  - isGeneratedBy – specifies the DataTableMap that generates the set of values for plotting. See the section on DataMaps.

  MappedDataTables Restrictions (Necessary Conditions):

  - The value of the isGeneratedBy property is a DataTableMap

[3] This could be extended to a script.

[4] This could be extended to a script.

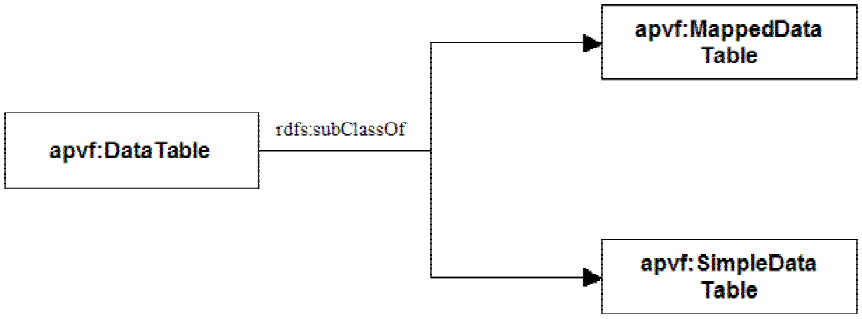


Figure 22: DataTable subclasses

3.8 DomainSpecificIndirection

A DomainSpecificIndirection is a group of options or choices associated with a property of a class. We use this information as a runtime lookup of alternatives and/or valid assignment based upon attributes of the runtime instance object.

The subclasses define one or more properties for the class, each of which has the same object type restriction on their range. This set of properties now makes up the choices or enumeration of values for the parent class. The choices are identified by property name, and their values are returned.

Subclasses of DomainSpecificIndirection:

- *TableCellDecoratorOptions* are the set of possible TableCellDecorator objects (see 3.9) that can be applied to a TableColumnInfoObject. The properties of the specific TableCellDecoratorOptions class are the names of the decorators that can be applied, and their value points to TableCellDecorator individuals defined. TableMappingFunctions use this property to assign the name of a decorator to apply to individual cells of a table.

3.9 TableCellDecorator

A TableCellDecorator is a way of formatting or using some sort of visual differentiator when displaying a specific cell in a table. The ontology supports the definition of visual effects and supplying DataMaps with threshold functions to select the appropriate visual effect at runtime by associating TableCellDecoratorOptions and TableCellDecorators with specific columns in a table as follows:

- Column properties include the hasOptions property. This property specifies a TableCellDecoratorOption to apply to the column.
- A TableCellDecoratorOption can define a decorator for the column-heading cell, a default decorator to be applied to each cell in the column, and any number of other decorators that should be applied to cells that meet certain criteria
- A TableCellDecorator defines the look and feel for a decoration.

TableCellDecoratorOption Properties:

- ColumnHeading – specifies the TableCellDecorator to apply to the column-heading cell.
- Default – specifies the TableCellDecorator to apply to each cell in the column (excluding the heading cell).

Figure 23: TableCellDecoratorOption properties

TableCellDecorator Properties:

- CellBackground – the background color for the cell.
- Font – the font type to apply to any text in the cell.
- FontBlink – whether or not any text in the cell should blink.
- FontBold – whether or not any text in the cell should be bolded.
- FontEmphasis – whether or not any text in the cell should be emphasized.
- FontItalic – whether or not any text in the cell should be italicized.
- FontUnderline – whether or not any text in the cell should be underlined.
- FontStrikethrough– whether or not any text in the cell should be stroked out.
- FontSuperscript – whether or not any text in the cell should be superscript.
- FontSubscript – whether or not any text in the cell should be subscript.
- FontSize – the size of the font.
- Icon – the URL of an image to be used in place of text for a cell.

If there is to be a table with cell decorations applied to one or more columns, the domain-specific ontology is defined with a TableColumnInfoObject for each column containing decoration. The TableColumnInfoObject is characterized by a set of decorators to potentially apply to individual cells of that column. The table itself will have a mapping function that contains the logic to evaluate the contents of the table cells and identify which cell decorator (if any) should be applied to the cell.

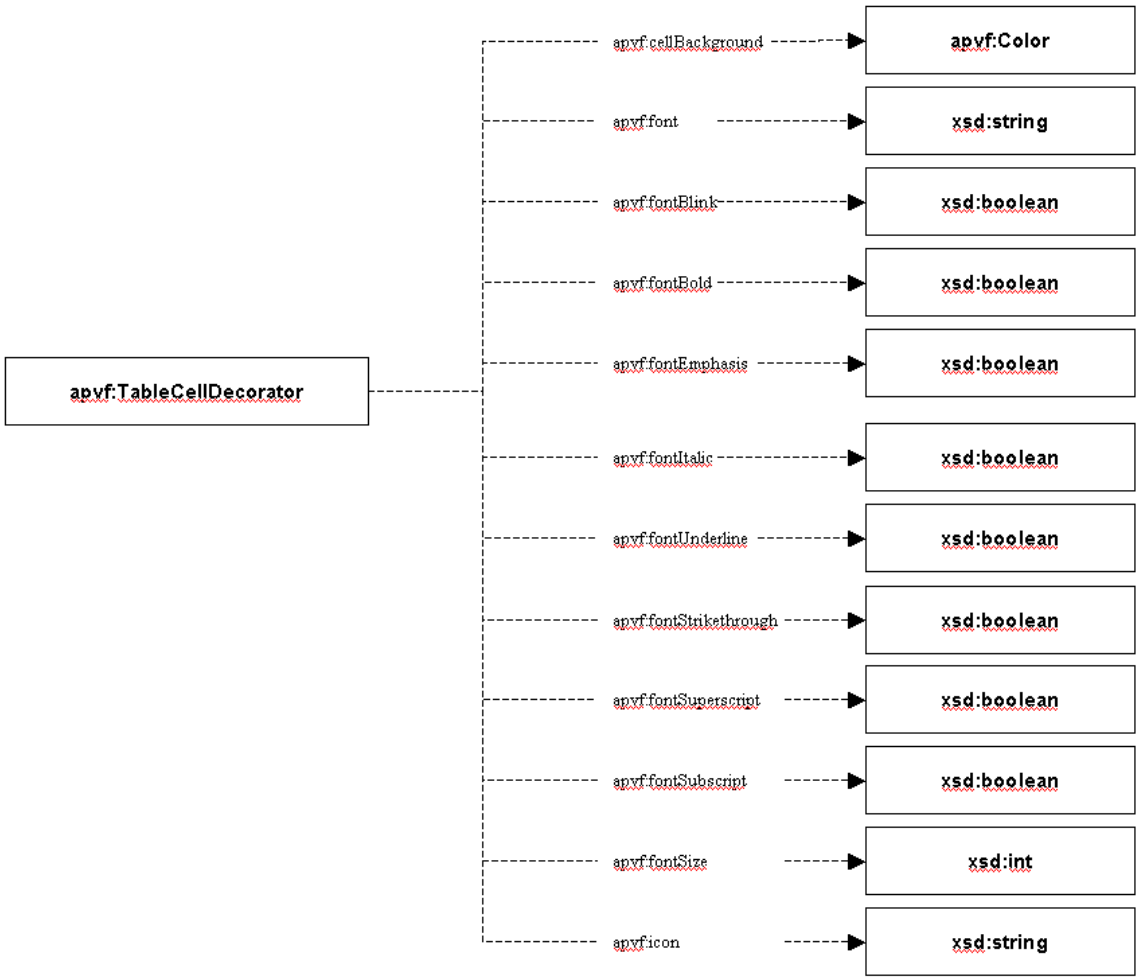


Figure 24: TableCellDecorator properties

3.10 DataMap

A DataMap defines how to translate a specific work domain data set represented by WorkDomainInformationObject into a format that presentation objects can render. We currently have two subclasses of DataMap: DataSeriesMap, which translates data sources to data series for plotting in a graph, and DataTableMap, which translates data sources to an array suitable for rendering in a table.

DataSeriesMap:

A DataSeriesMap is a custom transformation routine that implements a specified java interface supplied by the application designer. It is defined by the hasMappingFunction property of the Data Series Map.

DataSeriesMap properties:

- SeriesDelimeterColumn
- XMinColumn

- XMaxColumn
- YMinColumn
- YMaxColumn
- Encodes
- HasMappingFunction
- CandleWidth
- StartDate
- EndDate
- StartValue

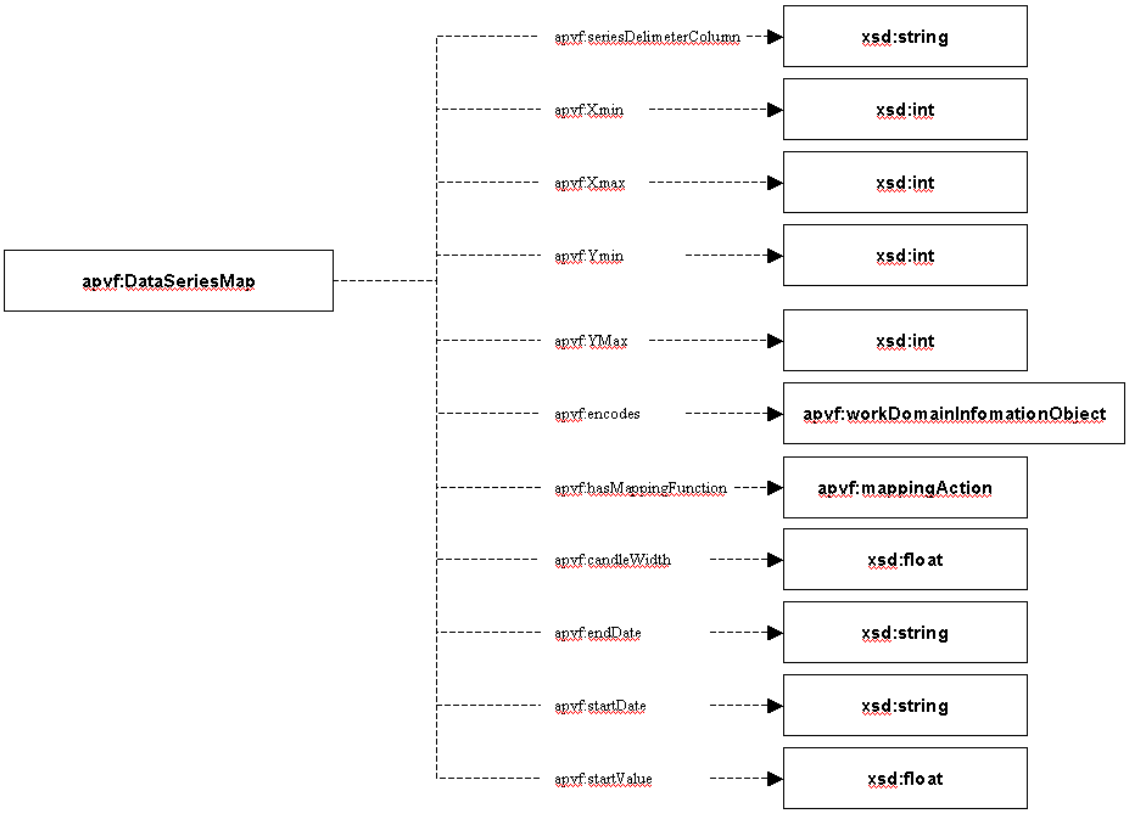


Figure 25: DataSeriesMap properties

ColumnOrderDataSeriesMap versus RowOrderDataSeriesMap:

A data series is a set of (xmin, xmax, ymin, ymax) values that are used for plotting a series in a graph. Typically the source data set will be an array of values, such that each row in the source data set provides a new (xmin, xmax, ymin, ymax) tuple to the plotted series. We call this type of mapping row order mappings. Row order mappings may

generate multiple data series, in which case there will be some column in the source data such that all the points in each generated data series share a common value from this column.

Column order data series use the data source to generate a new data series (for plotting) for each column in the source data.

Note that we assume the following:

- The source data are sorted primarily by the series delimiter values, if there is one
- The source data are otherwise sorted appropriately to produce desired results
- We will rarely mix and match row order and column order dataseries in the same graph.

DataTableMap

A DataTableMap defines how to translate source data to a two-dimensional array suitable for rendering in a table presentation object. A DataTableMap encodes (ObjectProperty apvf:encodes) a data source and uses the property hasMappingFunction to specify a custom transformation routine, which has a java interface and will be supplied by the domain application designer.

A data table mapping function produces a two-dimensional array of information, in row/column dimension format. Each element of the two-dimensional array is itself an array of size two. The first value represents the data value for the specified cell in the table. The second value represents the name of a TableCellDecorator that should be applied the specified cell at the time of rendering.

3.11 Highlight Region

DataTables and Graphs can have highlight regions, which are areas of information that are visually distinguished. The class HighlightRegion has the composite parameter Region (through ObjectProperty hasRegion), which has the composite parameter Slice with Parameters loThreshold and hiThreshold. Highlight regions can be independent or shared across multiple presentation objects. Highlight regions are shared by creating (transparent to the user) a SharedCompositeParameterSet (see Section 3.3) that has as members the HighlightRegions on the various presentation objects. A SharedCompositeParameterSet has a shared parameter as a property, which, in the case of shared highlight regions, is the Region. Each member of a SharedCompositeParameterSet no longer has its own value for the parameter identified by the shared set; instead, the ACUITy Controller uses the value of the shared parameter associated with the set as the value for each individual member. This means that a single Region can be defined for multiple highlight regions across multiple presentation objects. The attributes of the Region (such as color and area) can be modified and each of the set members will use the updated value.

3.12 DocumentObject

A DocumentObject is a subclass of PresentationObject that presents a resource referenced via a universal resource location (URL) to the user. A DocumentObject can either be rendered as a link to the referenced resources, or connect to and render the referenced resource in its native format.

Subclasses of DocumentObject

- ExternalWebApplication – a web-based application external to a domain-specific application instantiated within the ACUITy architecture. It has additional properties that capture the current URL, which is updated as the user navigates the external site. The effect of changes in the external web application (e.g. current URL) is captured by the hasEffect property (see Section 3.13).

DocumentObject Properties:

- DisplayText –the link text to display.
- encodes – the URL of the resource to lookup and fetch.
- hasPresentationNature – one of either ExternalResource or DocumentLink, defining whether the referenced resource is rendered via connecting to it, or if a link pointing to the location of the resource is rendered.

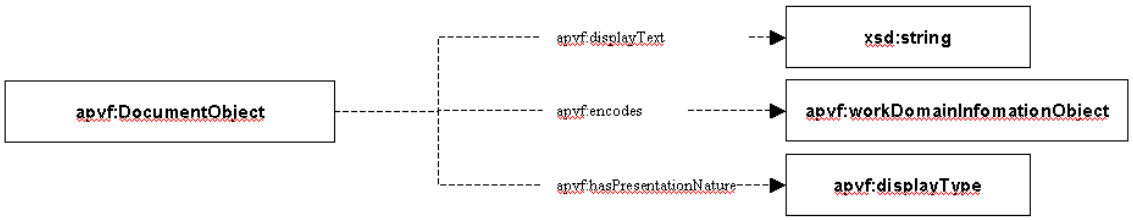


Figure 26: DocumentObject properties

3.13 InteractionObject

An InteractionObject is type of PresentationObject that collects input from the user.

Subclasses of InteractionObject:

- FreeFormInputObject – a text input box.
- SelectionListPresentation – a set of selectable choices.
- FormInputObject – a set of synchronized input and display objects (a form).
- HiddenObject – associates effects with cells in a table (e.g. hyperlinks to other presentation objects).

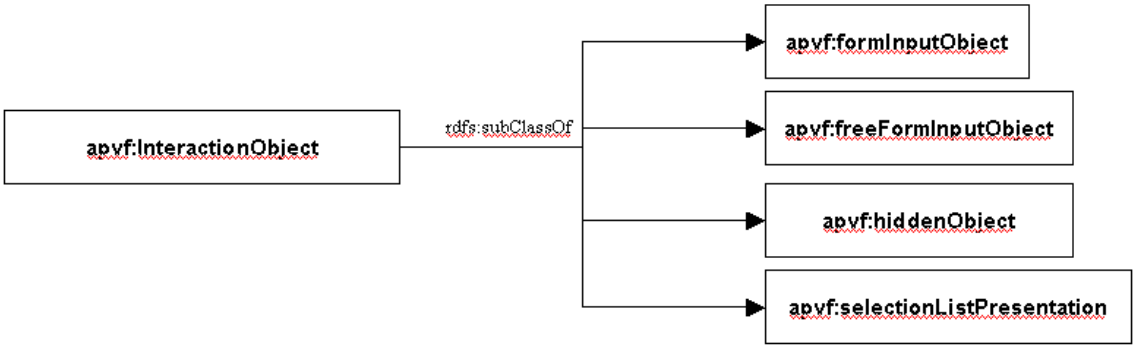


Figure 27: InteractionObject subclasses

InteractionObject Properties:

- hasEffect – defines the effect of submitting a value (or values) through the interaction object. The range of this property is Script (see section 3.14). Upon submission, the contents of the interaction object are passed on to the ACUITy Controller, which updates the ontology using the associated Script instance.
- orientation – (optional) the orientation, 0 for horizontal or 1 for vertical layout
- validateContent – (optional) true if the dependency of the content of the interaction object on anchor values, etc., is to be tracked and the content validated whenever a dependency changes.
- hasOwnSubmit – a Boolean property that indicates whether this interaction object should have its own submit button rendered in close proximity to it in the interface. If false, then submitting the value for the interaction object may depend on other interface objects having a submission mechanism.
- hasPresentationNature – the type of input control to be rendered (ListBox, DropdownList, Radio button, TextBox, TextArea, etc.)

SelectionListPresentation Properties:

- submitsOnSelect – a flag that indicates whether a submission should automatically occur after the user has made a selection
- numChoicesVisible – the number of possible choices that should be viewable at any one time within the interface
- hasPossibleAnswerGenerator (optional) – a Script to be used to generate the set of possible answers to be displayed in the selection list (see Scripts below)
- hasOnDemandAction (optional) – one or more Scripts to be included in the list of possible answers as user-defined actions

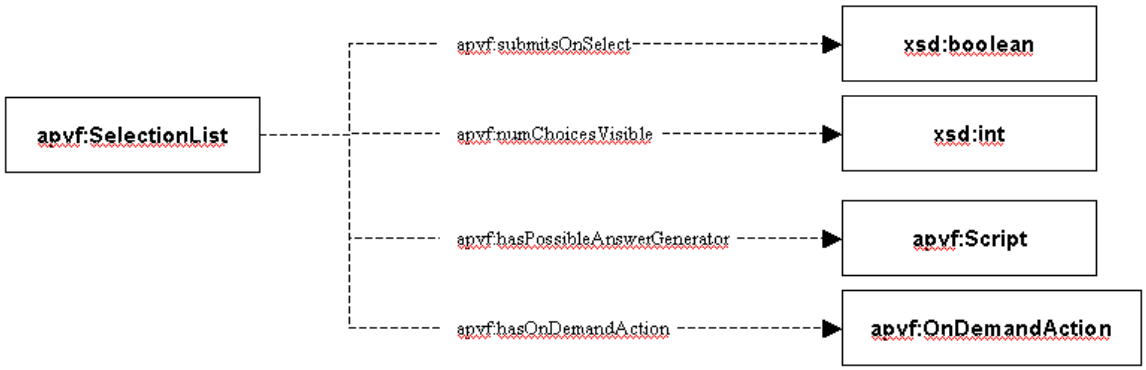


Figure 28: SelectionListPresentation properties

3.14 Script

A Script is an abstract thing that represents a sequence of instructions. Scripts include RDQL queries and xRDQL statements, which interact with the content of the ontology, stored procedures and SQL statements, which interact with relational databases, and ACUITy Controller Actions, which are implemented in Java code supplied by the application modeler. Scripts also include the simpler concepts of a SingleValue (a string value) and a ValueList, a set of comma-separated string values. In this section we'll discuss each type of script in more detail.

All scripts may have an optional string property, returnOnFailure, which, when present, specifies what the ACUITy Controller should return as the script value if the script execution fails. Otherwise the ACUITy Controller will throw an exception on failure.

Anchors

There is only so much that can be specified in the ontology at design time. Some things will not be known until the time the application is run. For example, the current frame, the focus vantage, and the current user will not be known at the time of design. An anchor is a placeholder specified at design time for a run-time value. At run-time the anchor is replaced with its actual value.

While most anchors reference by name a script which returns a value, there are several "bootstrapping" or predefined anchors in the ACUITy Controller. These are shown in the table below.

Anchor Identifier	Will be Replaced With
currentAC or self	The URI of the Individual which is the current instance of AcuityController
currentFrame	The URI of the Individual which is the current Frame associated ("hasFrame") with the current AcuityController
currentUser	The URI of the Individual which corresponds to the current user

focusVantage	The URI of the Individual Vantage which has the focus
currentIaO	The URI of the individual Interaction Object and for which an answer is being processed
useAnswer	The URI of the Individual which has been designated in the interaction as the answer
now	The current date/time
hostName	The canonical name, or IP address, of the host machine

Note that the syntax for de-referencing an anchor is to place ${...} around the name of the anchor in the referencing script. The anchor will be replaced with the URI of the individual(s) matching the anchor query. If only the local name is desired, the syntax is to place $# instead of just $ in front of the bracketed anchor name. Anchors can be used in any kind of script. First all anchors in the script are replaced with their values and, if this is successful, the containing script is then executed.

Extended RDQL (xRDQL Statements)

RDQL is a query language operating over RDF triples (W3C submission of 9 January 2004). A script written in RDQL can extract information from the ontology. Interaction anchors are a type of script with an associated RDQL query string property.

RDQL only has the ability to select information from an ontology; it does not include the ability to update or delete. However, we need a scripting language for ontology interaction that is capable of updating the ontology as well as querying it. While the W3C is working to create a standard ontology query language (SPARQL), current specifications do not include update capability. Therefore we have developed Extended RDQL (xRDQL), with update and delete capability. In addition, xRDQL allows multiple RDQL statements to be sequenced together, separated by semicolons[5]. Therefore an xRDQL script can be thought of as a set of selections, insertions, deletions, and/or updates of RDF triples in the application's ontology. Of course normally only the instance namespace (aBox) can be modified.

xRDQL extends RDQL in several ways and has the following syntax:

- INSERT (<subject>, <predicate>, <object>)[, (<subject2>, <predicate2>, <object2>)[,...]]

[5] Note that an xRDQL statement may also be embedded within another statement, e.g.,

UPDATE (<subject>, <predicate>, [SELECT ?o WHERE (<subject2>, <predicate2>, ?o)])

Each value returned by the SELECT statement will be assigned as the object of the update subject and predicate in an RDF triple. The square brackets are part of the syntax and identify the presence of an embedded statement.

- DELETE (<subject>, <predicate>, <object>)[, (<subject2>, <predicate2>, <object2>)[,...]]
- UPDATE (<subject>, <predicate>, <object>)[, (<subject2>, <predicate2>, <object2>)[,...]]

where the square brackets indicate optional additional triples and are NOT part of the syntax.

Some assumptions are required to implement these additional operations.

1. An INSERT will generally add new triples. However, to make use easier (require less knowledge on the part of the author of a statement) an INSERT of a triple that would result in a cardinality violation will have the effect, where possible, of doing an UPDATE. (See UPDATE below.)

2. An UPDATE will only succeed where there is no triple or where there is a single triple already existing that matches the UPDATE pattern. The first case is supported to require less knowledge on the part of the statement author—the statement is treated as an INSERT. In the case of an existing triple, that triple will be removed and a new triple added with the new information. If there are multiple triples that match the pattern, which one to update is ambiguous and an AcuityException will be thrown.

3. A DELETE maps to the Jena removeAll(Resource s, Property p, RDFNode o), which removes all matching statements and which allows one or more of the arguments to be null to indicate that any value of that argument will match. Rather than "null", an unbound variable, e.g., ?x, may be placed as the unconstrained argument.

4. Any subject, predicate, or object in a triple of an INSERT, DELETE, or UPDATE may contain an embedded RDQL SELECT statement. Since the SELECT itself may have any number of comma-separated triples, it is wrapped in square brackets to make its extent easier to parse. If the RDQL SELECT statement returns multiple values, the INSERT, DELETE, or UPDATE will be done only if doing so does not violate restrictions, e.g., cardinality.

5. Note that the SELECT used for the subject, predicate, or object of the triple, as described in #4 above and which is enclosed in square brackets, can be a composite SELECT which adds or removes values by subsequent SELECT statements. Such an aggregate SELECT is of the form:

   [SELECT ... PLUS SELECT ... MINUS SELECT ...]

   This is supported because the limitations of the RDQL SELECT make it otherwise difficult to combine and/or reduce a set of possible answers, e.g., provide a list of possible answers which are all of the POs in a Frame except those POs in the current Vantage. The variable(s) used in each SELECT must be the same and in the same order or an exception will be thrown.

SQLStatement

A SQLStatement has two DatatypeProperties: dbSQLString and defaultRowsetFieldIdentifier. The former identifies the SQL statement to be executed and the latter identifies which field in a return row set to use for returned values of the Script if it is referenced in another Script. They also have the ObjectProperty hasDBConnection, which identifies how to connect to the data source.

StoredProcedure

Like SQLStatement, StoredProcedure has an associated data source (hasDBConnection) and a defaultRowsetIdentifier. In addition, it has the DatatypeProperties (1) defaultOutputIndex, which identifies which stored procedure output (the return value being index zero) to use as the value of the script when referenced by another script, and (2) storedProcedureSignature. The stored procedure signature has the following syntax:

[?<type> =] call procedureName ([?]<type> [= ${anchor} | constant] [,…])

where <type> is the datatype, e.g., varchar2, cursor, etc., and square brackets indicate optional information and are not part of the syntax

OnDemandAction

OnDemandAction is a subclass of AcuityControllerAction. It inherits properties specifying the Java package, class, and method. It also has the property hasArguments, whose value is a string identifying a set of comma separated values that are to be passed to the action as Java method arguments. These values can be anchors or other scripts. Finally, an OnDemandAction has the property includeAsMissingPropertyOption, which, if true, indicates that this OnDemandAction should be included as an item on the menu of an Interaction Object created for a missing property whose object must be of the class with which this on-demand action is associated (via the hasOnDemandAction property).

SingleValue

This script type has only one DatatypeProperty, hasValue, whose range is a string. The value of this property is returned when an Individual of this type is referenced.

ValueList

This script type is similar to the one above except that its only property, hasValueList, with range string, is a comma separated set of values. For example, to populate a Selection List Interaction Object with the menu options 1, 2, 3, 4, 5, the hasPossibleAnswerGenerator property might have as object an Individual of type ValueList with a hasValueList value of "1, 2, 3, 4, 5".

3.15 Work Domain Information Objects

A WorkDomainInformationObject represents information about things that exist in or are pertinent to the work domain. If the instances and/or properties of any WorkDomainInformationObject exist in a relational database, they can be made to appear

as though they are in the ontology by the presence of a hasDBInstancesDescriptor or a hasDBStatementsDescriptor, respectively. For example, a pseudo-instance of WDIO (e.g. Fleet) could be given these properties in order to retrieve the actual instances and/or properties from an external data store.

Work Domain InformationObject properties:

- hasDBInstancesDescriptor -- (optional) relates a WDIO pseudo-instance to a DBInstancesDescriptor (see ACUITy Information Objects) when instance data is to be retrieved from a remote relational database source
- hasDBStatementsDescriptor – (optional) relates a WDIO pseudo-instance to a DBStatementsDescriptor (see ACUITy Information Objects) when statements (RDF triples) are to be retrieved from a remote relational database source

It should be noted that certain types of scripts, e.g., SQL statements and stored procedures, behave much like WDIOs but, for ease of implementation, they are not subclasses of WDIOs.

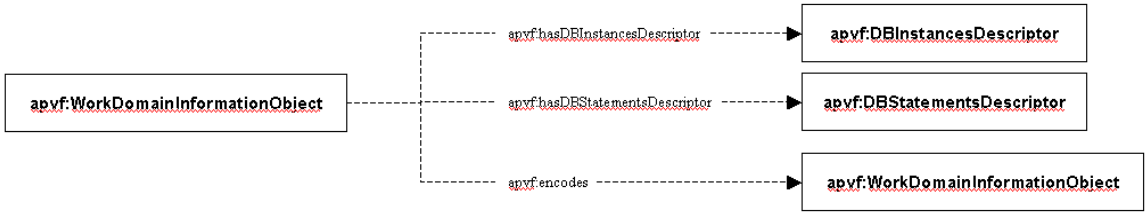


Figure 29: WorkDomainInformationObject properties

3.16 AcuityInformationObjects

An AcuityInformationObject represents special purpose information needed by the ACUITy Controller in order to perform certain tasks, such as processing data, maintaining user session, etc. For example, DBConnection is an AcuityInformationObject that represents information about how to access legacy databases.

DBConnection properties:

- DBConnectionString – the URL of a relational database.
- DBDriverString – the database driver to use perform operations on the database.
- LoginString – the user authentication information to use to connect to the database.
- PasswordString – the user authentication information to use to connect to the database.

A DBInstancesDescriptor represents a set of properties that identifies the elements necessary to query an external DB for instances of a class, i.e. “select <dbInstanceField> from <dbTable> where <dbClassField> = ‘classID>”, the “where” clause being optional.

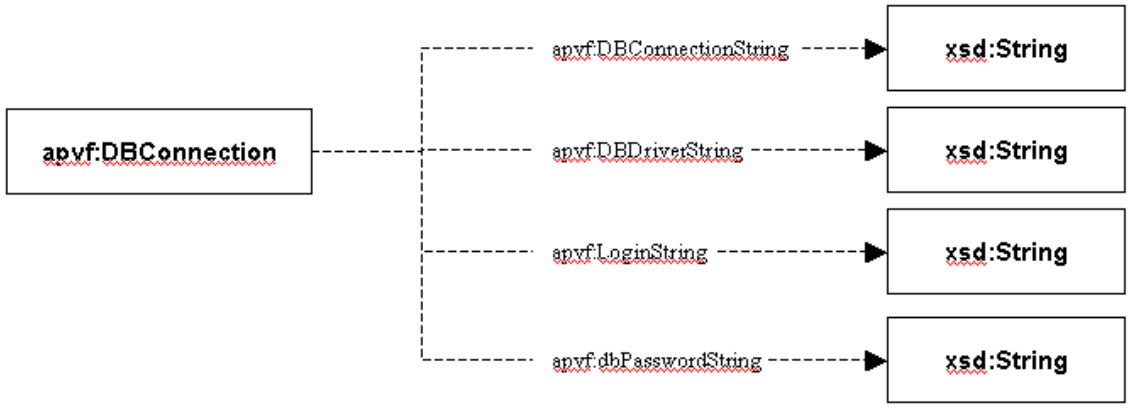


Figure 30: DBConnection properties

DBInstancesDescriptors properties:

- hasDBConnection – associates the connection information with the DBInstancesDescriptor.
- dbClassField – (optional) the name of the field in the ResultSet returned from the database which gives the class to which the instance in the row belongs.
- dbClassID – (optional) the value which the DbClassField must have to be included as an instance.
- dbInstanceField – the name of the field in the ResultSet returned from the database which contains the instance names.
- dbTable – the table in the database in which instance data is stored.
- dbInstanceStoredProcOutputIndex – (optional) only used for stored procedures; the index of the output to be used for the instance data—0 indicates the returned value of the stored procedure, 1 the first output argument, 2 the second output argument, etc.

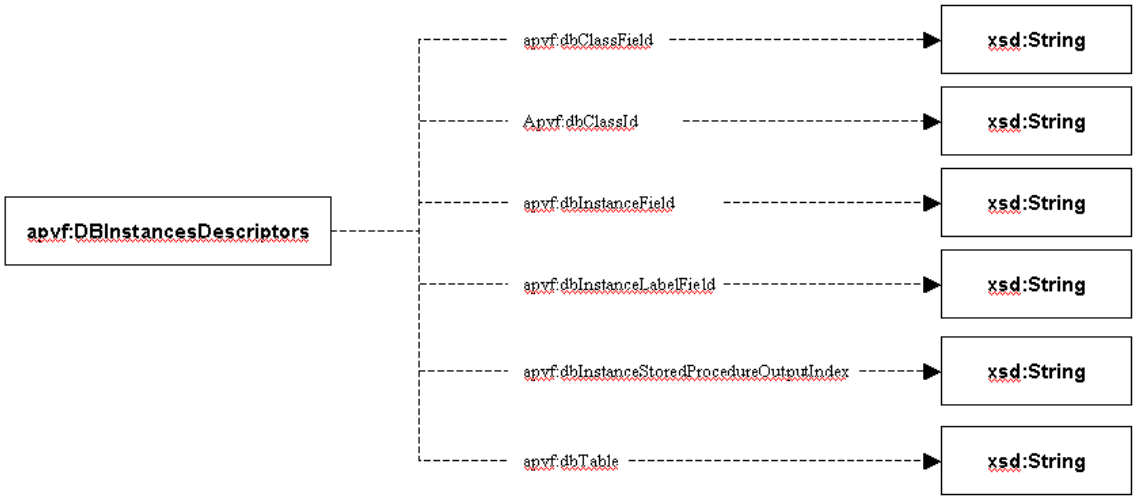


Figure 31: DBInstancesDescriptors properties

A DBStatementsDescriptor represents a set of properties that identifies the elements necessary to query an external db for the properties of an instance. i.e. “select <dbSubjectFieldName>,<dbObjectFieldName> from <dbTable> where <dbSubjectFieldName> = ‘<SubjectInstID>’” where <SubjectInstID> is the name of the individual with which this DBStatementsDescriptor is associated. The optional dbPredURI can be used to create Jena Statements; if one is not present, one will be created from the other properties of this instance.

DBStatementsDescriptors properties:

- dbObjFieldName – the name of the field containing the value of the property.
- dbPredURI – (optional) the pseudo-URI for Jena-style statements.
- dbSubjFieldName – the name of the field containing the subject of the retrieved statements.
- dbTable – the name of the database table containing the subject and value information.

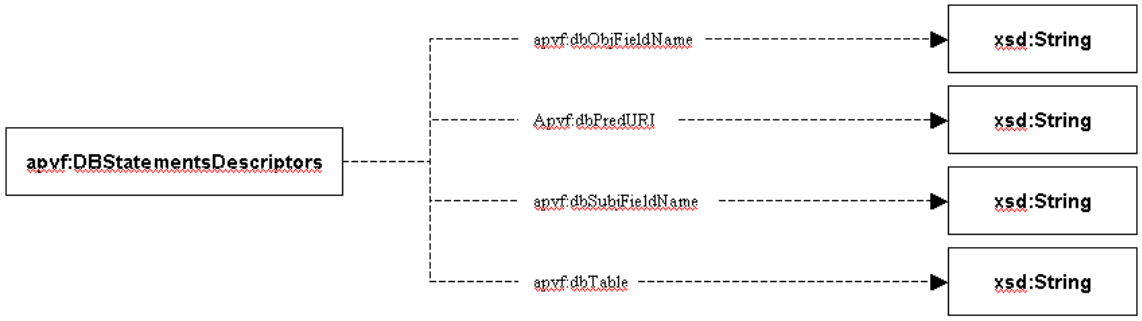


Figure 32: DBStatementsDescriptors properties

3.17 Specified and Learned Defaults

OWL DL (Description Logics) does not currently support default values. While we may wish to set a default value for a given property for new Individuals of a class, Properties cannot be reified in OWL DL such that they can have their own properties.[6] Furthermore, this approach would make the default value information visible to other OWL reasoners and, since there is currently no standard, might affect their behavior in undesirable ways.

We implement default values through the use of OWL annotation properties. In particular, we use the annotation rdfs:seeAlso, which “is used to indicate a resource that might provide additional information about the subject resource” [Brickley, et al., 2004]. In this case, we are providing additional information for use by the ACUITy Controller. We create a refining subclass called DefaultValue with subclasses ObjectDefault, IntegerDefault, BooleanDefault, etc. The subclasses have the properties has

[6] Properties can also be reified as Classes in OWL Full. However, even if making the ontology OWL Full were acceptable, the current tools for ontology editing, such as the Protégé OWL plugin, are geared toward OWL DL ontology development.

objectDefault, hasIntDefault, hasBooleanDefault, etc., respectively, with range Individual, xsd:int, xsd:Boolean, etc. These associations allow the specification of the value of the default. The default may be associated either as the rdfs:seeAlso of a property, in which case all instances of the property have this default, or as the rdfs:seeAlso of a class, in which case only the instances of the class have this default value for the property.

Note that one undesirable characteristic of this approach is that the appliesToPropertyWithName value is not a direct link to the actual property to receive the default value, but only an xsd:string which must match the property's name. Consequently, if the name of the property is changed, this string value must be edited as well to keep it synchronized with the actual property name. Of course, when it is sufficient to associate a default value with the property alone – the default will be the same regardless of the class to which it is applied – the rdfs:seeAlso annotation can be applied to the property instead of the class and appliesToPropertyWithName is not used.

All instances of DefaultValue may have the additional DatatypeProperties thresholdFrequency and/or minimumSampleSize. If the thresholdFrequency property is present, it indicates that the ACUITy Controller should look at the historical instance data to see if some value has occurred frequently enough to be used as the learned default. If so, this learned default would be used instead of the specified default. The threshold frequency is currently expressed as a decimal fraction, meaning that the number of found instances must be greater than or equal to that percentage of the total population. If the miniumSampleSize property is present, it indicates a minimum number of observations necessary before the learned default will be used. For example if minimumSampleSize is 20, then at least 20 historical instances must be in the histogram before the learned default will be used, assuming the threshold is met. If minimumSampleSize is specified but thresholdFrequency is not, then as soon as the sample size requirement is met the most common historical value will be used as the default regardless of frequency. In the case of ties, the choice among the top contenders will be arbitrary.

The histogram of values, from which a learned default value of a Property for a given Individual may be derived, is generated by an RDQL query that returns all of the values that the Property has been given for all Individuals of the same type.. To handle user-specific preferences, we must restrict the learned value to those instances of the same type that were created by the current user. A more general user-based learned value could look at preferences of subsets of users of the same type. For example, in the absence of sufficient data for a new user who is a maintenance planner, we could query the preferences of other maintenance planners (e.g., as opposed to including the preferences of field engineers).

4 Summary and Future Directions

We have developed a functioning ontology-driven architecture that embodies several innovations, including legacy data access, xRDQL and the execution of procedural actions based on a declarative OWL ontology. Specifying and learning defaults also represents a significant accomplishment, as does formalizing the Problem-Vantage-Frame approach as a framework for modeling information display and interaction.

However, there are still many questions open for future research and development. First, we would like to expand the basic capabilities of the APVF model architecture to include the following:

- Process flows,
- Responding to notifications (from within and without the ontology),
- Switching between representational formats (eg. converting tables to graphs, etc.),
- Additional visualization/interaction mechanisms such as multi-media interactions (e.g. audio, images, animation), and tactile and immersive interactions, and
- Problem representations at a lower level of granularity and models of solutions and/or problem-solving.

Second, we plan to incorporate more advanced automated reasoning for decision support, including:

- Identifying and disseminating peer group preferences such as specifying and grouping users by roles,
- Mechanisms to assess the success or failure of a problem/problem-solving model,
- Identifing and learning new problem/problem-solving models from user adaptations and actions, and
- Notifying appropriate user groups of model changes.

Third, we plan to integrate this architecture with the emerging Service Oriented Architecture.

Finally, we foresee the need for a user-friendly, visual ontology editor, specifically designed to edit domain specific ontologies using the PVF model (we currently browse and edit our ontology using the OWL Plug-in to Protégé).

As we continue our research and as the architecture is exercised to produce more applications, we expect to identify additional areas for future development.

5 Acknowledgements

We would like to acknowledge the contributions of Lt. Mona Stilson, Dr. Bob Eggleston, Jim McManus and Capt. Tony Aultman from Wright Patterson Air Force Base, Human Effectiveness Directorate. The principles of Work-Centered Support Systems and Work-Centered Design concepts promoted by Dr. Eggelston were our initial inspiration for the ontology described in the paper, and the WPAFB team's consistent encouragement to focus on the work-centered framework for designing novel human-computer interactions helped us maintain and validate the direction of our research.

We would also like to acknowledge the contributions of Robert Mudge and Robert Norton, Lockheed Martin Simulation and Training Systems, whose domain expertise provided a real-world sounding board for evaluating and refining the ideas outlined here.

Appendix A. More Details about the ACUITy Controller

A.1 Introduction

The AcuityController is a Java class that, along with other supporting classes, provides an application programmers interface (API) to a semantic model of a Problem-Vantage-Frame implementation of Work Centered Support (Eggleston and Whitaker 2002, Eggleston et al. 2000). The semantic model is ontology-based and is intended to be robust and extensible, capable of supporting adaptive user-interfaces. While the AcuityController is largely domain-independent, it is designed to be extended (sub-classed) if needed to create domain-specific interfaces. In general, domain-specific knowledge is captured in the ontology and a customized controller is not needed. The Problem-Vantage-Frame knowledge is expressed in the Web Ontology Language (OWL) and it is assumed that domain-specific knowledge will be similarly expressed. This document describes the overall architecture and design approach used to create the AcuityController along with selected functionality.

A.2 A Hierarchy of Ontologies

The AcuityController supports hierarchies of ontologies through use of the OWL import functionality. For example, a very basic classification of things that exist, e.g., physical versus abstract, is represented in the ACUITy upper-level ontology (namespace URI *http://research.ge.com/Acuity/aulo.owl*, abbreviated as prefix *aulo*), which is then extended to include Problem/Vantage/Frame concepts in the ACUITy Problem-Vantage-Frame ontology (namespace *http://research.ge.com/Acuity/apvf.owl*, abbreviated as prefix *apvf*). The latter ontology uses the "owl:imports" construct to reference the former. Similarly, a domain-specific ontology can be placed in one or more owl files and can then import these domain-independent ontologies. An ontology can use any number of "owl:imports" to include existing ontologies, which can be used and/or extended in the importing ontology. Note that even if an application uses the relational database ontology storage option (described below), it will probably be the case that the ontology will be designed in individual owl files and subsequently be moved to the database storage.

Note that while OWL ontologies, including those that are imported, have a public namespace, it may be desirable to actually load them from a different URI. Jena (Jena), upon which the AcuityController is built and which is used to load the ontologies, allows a mapping between public and actual URIs to be specified in an ontology policy file, usually named *ont-policy.rdf*.

A.3 Instance Data and Persistent Memory

Any OWL ontology can include instances as well as the declarations of classes and properties (relations). In Description Logics (DL) nomenclature, the abstract class and property declarations are the tbox (terminology) and the instance data is the abox (assertions). It is our objective to distinguish between the extension of an ontology by defining new classes and/or properties (new abstractions) and the extension of an ontology by including (remembering) new instances. The former is at the heart of an

extensible user-interface capable of handling new kinds of problems while the latter is a simpler form of learning useful in achieving adaptive behavior. For example, each time a particular user (an instance of Person) chooses to see a particular piece of information related to a particular problem, remembering the problem and the information used (instances) allows the AcuityController to adapt future displays for this particular user to make the information the user is most likely to consider relevant more prominent. Since the ontology captures the generalizations, e.g., owl:Classes, to which instances belong, it is often more useful to aggregate information to the Class level in order to, for example, provide default information for a new instance of a class of problems based on the remembered history of other instances of that class.

In implementing an instance memory, a potential conflict of objectives is encountered. It may be desirable to load a domain's ontology hierarchy, as described above, as URIs from a web server. However, since it is not generally possible to write the instance data acquired in the course of a session back to a URI, a file storage medium may be desirable. The problem is resolved by using the "owl:imports" tab described in the previous section. The instance data (abox), which may be loaded from a local file, imports the abstract declarations (tbox) from a server. At the end of the session, the instance data may be written back to the file system ready for use in subsequent sessions by calling the AcuityController *save* method. In this way, the core knowledge can be loaded from a central repository (web server) while each application can use its own instance memory. The core knowledge can be extended without affecting instance memory as long as changes are not made which render existing instances inconsistent with the terminology. Note that a local instance cache of this type must be initialized in some way as described below.

Alternatively, a database backend can be used to store both tbox and abox information. In general, better performance for small ontologies is achieved with the file-based approach described above. The AcuityController method *convertModelToDatabaseModel* allows an ontology to be converted to a database model using Jena functionality. Existing instance (abox) content is converted and can be augmented with additional instance data in the database model in a persistent manner. An alternate AcuityController constructor allows the database model to be loaded as the active model. Note that unlike the file-based instance storage described above, where the *save* method must be called to persist instance data new in this session, when db-based persistence is used the *save* method must be called to delete any temporary instance data, which is not to be persisted, from the database.

A.4 Initialization of an AcuityController

Consistent with the above design considerations, the following information must be provided to create an instance of and initialize an AcuityController. Where defaults can be provided, constructors are provided which use the default values noted. For constructor details see the AcuityController JavaDocs documentation.

If the model is loaded from an OWL file, either local or at a URI, the following information is needed:

1. The Jena ontology policy file. If this mapping file is not specified (a constructor is used without this argument) or is null, all imported ontologies must actually be found at their public URIs.

2. The path and name of the local instance file persistent store. This can be unspecified or null if no instance data is to be persisted.

3. The URI of the ontology to be used for initialization if no content is found in the local instance file. This initialization content will be replicated, along with any new instance data, in the content written to the local instance file, if named, when *save* is called. Note that at a minimum the initialization file should specify the namespace of the instance data to be remembered and an owl:imports tag with the location of the tbox ontology. Of course this tbox can import other ontologies, etc. While this initialization ontology is not used when there is an existing instance store, it must be present for an instance store to be created and would not normally be null.

4. An OntModelSpec (a Jena class) identifying the reasoner (if any) to be used. By default the OntModelSpec.OWL_MEM_TRANS_INF (OWL in-memory transitive reasoner) is used.

5. The trace level for debug or informational output from the AcuityController and the underlying Jena library. By default, no tracing is enabled (0). The possible values of this flag are found in the AcuityConstants Java class.

If the model is loaded from a database (DB model, which provides instance persistence) the following information is needed:

4. The database ontology's public URI (namespace).

5. The database URI, e.g., "jdbc:mysql://<host>/<dbname>" where host and dbname are specified as part of the database instance creation and must be setup in advance.

6. The database user.

7. The database user's password.

8. The database type, e.g., "MySQL". This value is recognized by the Jena database backend.

9. The database driver Java class name, e.g., "com.mysql.jdbc.Driver".

10. An OntModelSpec (a Jena class) identifying the reasoner (if any) to be used. By default the OntModelSpec.OWL_MEM_TRANS_INF (OWL in-memory transitive reasoner) is used.

11. The trace level for debug or informational output from the AcuityController and the underlying Jena library. By default, no tracing is enabled (0). The possible values of this int flag are found in the AcuityConstants Java class.

A.5 Interaction with an Initialized AcuityController

Once initialized by loading the knowledge base (tbox) and local instance data (abox), the AcuityController API supports several different kinds of interaction. It is useful to remember the essential role of RDF triples in all OWL ontologies. An RDF triple can be thought of as a pseudo-English sentence in the form of (subject, predicate, object). In Jena terminology, an RDF triple is a Statement. For example, to tell the knowledge base that George is a Person (is of type Person), the information content of the tell expression (omitting namespace information) would be the Statement or triple:

(George, type, Person)

A.5.1 Namespace Considerations

However, in order to make the three parts of the RDF triple "machine understandable", each must actually be a fully qualified URI. Such a URI can be constructed from two parts; a namespace designator (URI followed by a "#") and a fragment identifier or local name within the namespace. For example, type in the triple above is really:

http://www.w3.org/1999/02/22-rdf-syntax-ns#type

If Person is defined in the namespace http://research.ge.com/Acuity/aulo.owl and George is an instance in namespace http://research.ge.com/Acuity/rbProto.owl, the complete RDF triple would be:

(http://research.ge.com/Acuity/rbProto.owl#George,
http://www.w3.org/1999/02/22-rdf-syntax-ns#type,
http://research.ge.com/Acuity/aulo.owl#Person)

A.5.1.1 AcuityConstants Namespace Named Static String

To facilitate the correct use of namespaces, the AcuityConstants class contains the namespace for the aulo.owl file (AcuityConstants.ULO_NS) and the apvf.owl file (AcuityConstants.PVF_NS), along with some other important and frequently referenced namespaces, as static Strings. Below are some examples of how to use these constants along with their actual values in parenthesis.

- AcuityConstants.RDF_NS (http://www.w3.org/1999/02/22-rdf-syntax-ns#)
- AcuityConstants.RDFS_NS (http://www.w3.org/2000/01/rdf-schema#)
- AcuityConstants.OWL_NS (http://www.w3.org/2002/07/owl#)

- AcuityConstants.TIME_ENTRY_NS (http://www.isi.edu/~pan/damltime/time-entry.owl#)

The AcuityConstants class also provides the static function, createNS(nsDesignator, fragID), to combine the namespace designator and the fragment identifier. This method can be used to create the argument to the Jena OntModel methods createResource, createProperty, createIndividual, etc. For example, if an AcuityController *ac* has been created which loads the rbProto.owl file, the Java code to create the Jena Resource for *George* is:

```
AcuityController ac = ...;
Resource cls =
ac.getResource(AcuityConstants.createNS(AcuityConstants.ULO_NS,
"Person"));    // find the Resource for the class Person
Resource r =
ac.createIndividual(AcuityConstants.createNS(AcuityConstants.ULO_NS,
"Person"),
AcuityConstants.createNS("http://research.ge.com/Acuity/rbProto.owl#",
"George"));    // create a new instance of Person
```

Domain-specific implementations can extend the AcuityConstants class with additional namespaces. For example, the ErpConstants class contains the following Java code:

```
public class ErpConstants extends AcuityConstants {
      public static final String ERP_KB_PUB =
"http://research.ge.com/Acuity/aerpData.owl";
}
```

In this way any extension of AcuityConstants, e.g., ErfConstants, can be used to reference any namespace defined either in the subclass or in the super class.

A.5.2.1 Namespace Prefixes

Alternatively, RDF files in general and OWL files more specifically allow the designation of a namespace prefix in the opening RDF tag. For example, one of the elements of this tag in the Acuity upper-level ontology file (aulo.owl) is:

```
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

This element defines the string "rdf" as the prefix for the namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#". This information allows one to abbreviate a reference to "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" as "rdf:type". Assuming the prefixes "rbproto" and "aulo", the RDF triple above becomes:

```
(rbproto:George, rdf:type, aulo:Person)
```

Whenever a Resource name is passed to a method in an AcuityController instance, if the name does not appear to be a complete URI, i.e., there is no "#" in the name, the

AcuityController will look to see if a prefix is provided. If there is a colon in the name, the portion of the name before the colon is used to check for a prefix in the ontology model's namespace prefix map. If it matches a known prefix, the prefix is replaced with the corresponding namespace. Therefore, the code segment shown above to demonstrate the use of static Strings in the class AcuityConstants could also be implemented as follows, assuming the prefixes shown have been defined:

```
AcuityController ac = ...;
Resource cls = ac.getResource("aulo:Person");                    // find the
Resource for the class Person
Resource r = ac.createIndividual("aulo:Person", "rbproto:George");     // create a
new instance of Person
```

A.5.3.1 The [Dubious] Assumption of Uniqueness

The AcuityController provides another means of assisting with namespace management. In many cases, the local name will be unique--the same local name will not be used in any referenced or imported ontologies used in the application. When this is the case, the AcuityController's getResourceURI(localName) method can be used explicitly. This method will find a namespace, assumed to be the only one, containing the local name and return the complete URI. Alternatively, the AcuityController will check a Resource name and if it appears to be neither a complete URI nor a prefix followed by a local name, an attempt to find a matching local name in the ontology. This of course will only work for existing Resources. Thus in the code example could be rewritten as:

```
AcuityController ac = ...;
Resource cls = ac.getResource( "Person");                    // find the
Resource for the class Person
Resource r = ac.createIndividual( "Person", "rbproto:George");     // create a new
instance of Person
```

The second argument of the *createIndividual* method call cannot be "George" because this local name does not yet exist. However, the class "Person" does and so that Resource can be found. Care should be taken that the assumption of uniqueness is met and continues to be met as an ontology evolves if this AcuityController feature is used.

A.5.4.1 Instance Data Namespace

Two AcuityController methods make use of the instance data namespace a little easier. The instance data namespace can always be retrieved from an instance of the AcuityController by calling *getInstanceDataNamespace*. A local name fragment in the instance data namespace can be converted to a URI by calling *mkInstanceDataURI*. The example above could be rewritten as:

```
AcuityController ac = ...;
Resource cls = ac.getResource( "Person");
// find the Resource for the class Person
```

```
Resource r = ac.createIndividual( "Person",
ac.mkInstanceDataURI("George"));      // create a new instance of Person
```

Note that all Individuals created by the AcuityController have the instance data namespace even though the owl:Class of which they are an instance may be in a different namespace.

A.5.2 Tell and Ask

Perhaps the most fundamental interaction with the AcuityController is to ask questions and tell new information. RDF triples are the primary expression used in either case.

For example, given an AcutiyController instance ac1, the RDF expression, without namespace information, to tell the knowledge base that George is the user of this instance would be:

```
(ac1, hasUser, George)
```

Of course addition of the correct namespace information is essential for achieving the desired behavior from the ontology. If *hasUser* is an Object property in the apvf.owl ontology and *AcuityController* is a subclass of the aulo.owl class *ArtificialAgent* with instance *ac1* defined in the rbProto.owl ontology, the fully qualified RDF triple for stating the above assertion is:

```
(http://research.ge.com/Acuity/rbProto.owl#ac1,
http://research.ge.com/Acuity/apvf.owl#hasUser,
http://research.ge.com/Acuity/rbProto.owl#George)
```

or, assuming the prefixes shown:

```
(rbproto:ac1, apvf:hasUser, rbProto:George)
```

The AcutiyController class provides a *tell* method for imparting new information to the knowledge base. This method requires three arguments: 1) a subject, 2) a predicate, and 3) an object. The following code segment illustrates use of the tell method to assert that George is the user of ac1, after first creating the necessary resources.

```
AcuityController a = ...;
Property p = a.getOntProperty("apvf:hasUser"))                    // get the
Property "hasUser"
Resource newPerson = a.createIndividual("aulo:Person", "rbproto:George"));    //
create a new Person instance named "George"
Individual acInst  = a.getAcuityControllerInstance( );             // the
Individual representing this instance of AcuityController
a.tell(acInst, p, newPerson);                                     // tell the
controller that the it hasUser George (the new person)
```

Note that the methods *tell* and *addStatement* are identical in function.

While telling is always in the form of the assertion of an RDF triple, asking has two forms. The first is through pattern matching. When any of the property (predicate), the subject, and the object of an RDF triple expression is set to *null*, pattern matching will be used to return all statements in the knowledge base, asserted or inferred, which match the query expression. For example, to ask for all of the individuals of *type Person*, one would make the following call:

```
AcuityController a = ...;
Iterator iter  = a.getMatchingStatements(null, "rdf:type", "aulo:Person");
while (iter.hasNext( )) {
   System.out.println(((Statement)iter.next( )).getSubject( ).getLocalName( ) + "
is a Person");
}
```

Note that the method *getMatchingStatements* is identical to the *ask* method. Thus to ask for all of the instances of things related in any way to the *AcutiyController* instance *ac1* (*ac1* as the subject), along with the kind of relationship:

```
AcuityController a = ...;
Iterator iter  = a.ask(a.getAcuityControllerInstance( ), null, null);
while (iter.hasNext( )) {
   Statement s = (Statement)iter.next( );
   System.out.println("ac1," +  s.getPredicate( ).getLocalName( ) + ", " +
s.getObject( ).toString( ));
}
```

The AcuityController also supports the SQL-like RDF query language RDQL. In RDQL, the two queries above (omitting namespace information) would be:

```
SELECT ?s WHERE (?s, <type>, <Person>)

SELECT ?p, ?s WHERE (<ac1>, ?p, ?o)
```

The real reason for using RDQL is to enable much more complex queries. For example, the following query in a genealogy ontology finds everyone's aunt :

```
SELECT ?c, ?a WHERE (?c,
<http://camfe.org/FamilyThreads/family.owl#hasParent>, ?p), (?p,
<http://camfe.org/FamilyThreads/family.owl#hasParent>, ?gp), (?gp,
<http://camfe.org/FamilyThreads/family.owl#hasChild>, ?a), (?a,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://camfe.org/FamilyThreads/family.owl#Woman>) AND ! (?a eq ?p)
```

This expression says find all of the children (c) who have a parent (p) who in turn have a parent (gp) who in turn have a child (a) who is a Woman and who is not a parent (p) of the child (c).

A.6 Problem-Vantage-Frame Ontology at First Glance

While our approach to implementing the Problem/Vantage/Frame approach is discussed in greater detail in a later section of this document, a brief overview may be helpful in understanding the material preceding that detail. The AcuityController as a concept (owl:Class) is found in the *apvf* ontology, and each client session will instantiate an instance of this class. The Frame class is the highest-level container of the user-interface--it roughly corresponds to a session. A Vantage is the next level user-interface class. A Vantage represents a particular point-of-view. Each Frame instance can contain one or more Vantage instances. The *apvf* ontology assumes that only one Vantage will be the focus of attention at any given time--the focus Vantage.

Besides Frames and Vantages, the *apvf* ontology identifies two other classes of user-interface objects. InteractionObjects display information to the user, either as a menu or a text input box, and allow the user to send an "answer" back to the AcuityController. DisplayObjects only display information without any interaction possibility. A Vantage or Frame can contain any number of InteractionObjects and DisplayObjects. Frame, Vantage, InteractionObject, and DisplayObject are all subclasses of PresentationObject. By ontological restriction, a Frame or Vantage class can be defined so that it must contain certain kinds of PresentationObjects within it.

A.7 Specifying Procedural Behavior using AcuityController Actions

Ontologies are very declarative in nature, meaning that they define what exists structurally but do not easily represent behavior. The AcuityController, in conjunction with the Problem/Vantage/Frame ontology (namespace URI *http://research.ge.com/Acuity/apvf.owl*), implements some useful behaviors. These behaviors are grouped in two categories: 1) missing Properties, and 2) new Individuals.

A.7.1 What is a Missing Property

Missing Properties is the name we give to an inconsistency between the necessary conditions imposed by Restrictions on an OWL Class and an actual instance (Individual) of that Class. For example, our ontology might define the Class Mother as a Woman with a someValuesFrom restriction on the Property hasChild, which in turn has as range the class Person. If the instance data declares that Jane is an instance of the Class Woman but upon querying the ontology we find that Jane has no hasChild Properties, we may conclude that Jane has a missing Property (hasChild) with range Person. In the Problem/Vantage/Frame ontology, a Frame has necessary conditions that require it to have a focus Vantage (*apvf:hasFocusVantage* Property) and a VantageSelectionObject related to the Frame by an *apvf:containsPresentationObject* Property.

A Class definition in an ontology may include a Restriction that a particular Property must have someValuesFrom a particular range Class, or it may include a Restriction that a particular Property has a cardinality greater than or equal to one or a minCardinality greater than or equal to one. When one or more of these Restrictions is included in the Class definition and an Individual of that Class exists but no Statement with that

Individual as subject and the particular Property as predicate is found in the ontology, we refer to the situation as the Individual having a "missing Property".

A.7.2 What to Do When an Expected Property is Missing

The missing Properties of a particular Individual or those of all Individuals in the ontology can be found by calling the AcuityController method getMissingProperties. The returned value will be null if no missing Properties are found or a List of InteractionContent structures (see JavaDocs for details). Each InteractionContent instance in the List identifies, among other things, the subject (Individual), the predicate (missing OntProperty), the type of Restriction encountered which implies the missing Property along with the Class or Classes to which the object of the missing Statement must belong (which we shall loosely call range), and cardinality information.

The AcuityController identifies three basic types of actions that can be taken in the event of a missing Property:

1. Automatically create a new Individual of the Class identified as the range of the missing Property. This behavior is accomplished by making the Property of the Restriction or cardinality constraint an rdfs:subProperty of apvf:autoInstantiateProperties. This will only work if the range of the restricted Property is a single Class. If the missing Property is sub typed in this way and the range includes multiple Classes an AcuityException will be thrown.

2. Execute a user-defined procedure expressed as a Java method on a Java class. The procedure is fully identified by an Individual of type apvf:MissingObjectPropertyAction, which has Properties identifying the package, class name, and method name. The implementation method must take four arguments of type AcuityController, IndividualImpl, ObjectPropertyImpl, and OntClassImpl, which will be, when passed at runtime, the Java instance of the AcuityController doing the processing, the subject Individual, the predicate Property, and the range Class of the missing ObjectProperty. The method may return a List containing the URIs of the Individuals which can be the object of a new ObjectProperty and/or the OnDemandActions which can be taken. OnDemandActions are discussed below.

3. Ask the user to identify an Individual to be the object of a new Statement with the Individual missing the Property as subject and the missing Property as predicate. If the class has a hasValue Restriction on Property hasAskUserRDQLValueFilter, the value (an RDQL query string) will be used to generate the List of possible answers, presumably a subset of the Individuals which are of the type(s) identified by the range. If there is no such hasValue Restriction on hasAskUserRDQLValueFilter, all Individuals that are of the type(s) specified by the range of the Property will be identified as possible answers by calling the AcuityController method *addPossibleAnswerURIs*. Any Property which is a subProperty of the Property apvf:askUserPropertiesIncludeCreateNew, e.g., apvf:hasFrame, will, when it the predicate of a missing Property which is to be asked, included the create new action by default in the list of possible answers. If

the special instance (Individual) of class apvf:OnDemandAction named apvf:createIndividualOfRangeClass (see below) is associated with the Class by a hasValue Restriction it will also be included in the List.

OnDemandAction, MissingPropertyAction, and InstanceCreationAction are each an owl:subClass of AcuityControllerAction in the *apvf* ontology. All AcuityControllerActions identify a Java static method in a specified package and class that implements the action. An OnDemandAction implementation method must have two arguments of type AcuityController and Object[], which will be this instance of the AcuityController followed by an array of arguments to the action. The second argument, if not an empty array, is expanded to form additional arguments to the method. The method has no return value (is void). OnDemandActions are associated with a Class as a hasValue Restriction on the Property apvf:hasOnDemandAction. Note that a special instance of OnDemandAction defined in the *apvf* ontology and known explicitly to the AcuityController is the Individual apvf:createIndividualOfRangeClass. This individual doesn't actually have any Java code associated with it but makes available as an OnDemandAction the same Individual creation functionality identified as the "automatic creation," described in the first item in the list above.

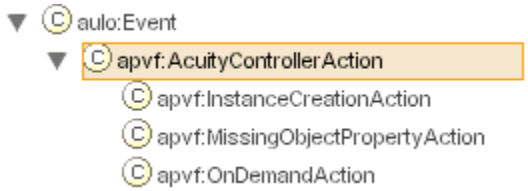
A.7.3 Action when a New Individual is Created

When a Class has a hasValue Restriction on the apvf:hasInstanceCreationAction (which will necessarily have a value which is an Individual of type apvf:InstanceCreationAction), the Class is given a behavior defined by this value. In particular, the behavior is invoked whenever a new Individual which is an instance of this Class is created.

Of course the creation of a new Individual can result in new missing Properties and so lead to additional behavior through the mechanisms described above.

A.7.4 Summary of Action Classes and Properties

The figure and table below shows the AcuityControllerAction Class and its three subclasses representing the three basic types of action supported along with usage information.



Class	Static Method Arguments and	Description

	return type	
InstanceCreationAction	1. This instance of AcuityController 2. the IndividualImpl newly created 3. the OntClassImpl of the Individual newly created Method is void (does not return anything)	The static method is called when the instance (Individual) of this class is the value of a hasValue Restriction on a hasInstanceCreationAction Property on a given Class of which a new instance (Individual) has just been created.
MissingObjectPropertyAction	1) this instance of AcuityController 2) the Individual which is the subject of the missing Statement 3) the OntProperty which is the missing Property 4) the OntClass which is the range of the object of the missing Statement. Method returns a List of possible object values and actions or null.	The static method is called when the instance (Individual) of this class is the value of a hasValue Restriction on a hasMissingObjectPropertyAction Property on a given Class which is the range of the object of a missing Property. Note that this method may be executed multiple times. In particular, execution will be triggered when the subject of the missing Statement is the target of a getMissingProperties call. It will also be triggered when the subject of the missing Statement has just been created. Note: if the method returns a non-null List, the MissingPropertyInfo instance passed back on a call to getMissingProperties will have the askUser flag set to true indicating missing Property is "askable."
OnDemandAction	1. this instance of AcuityController 2. an array of type Object[] (which can be null) containing any additional arguments to be passed to	The static method may be called by the client application by calling the AcuityController method processOnDemandAction(String actionURI, Object[] addlArgs). Alternatively, since OnDemandAction instances may be included in the list of possibilities for an askable missing Property, a call to processMissingPropertyAnswer(M

	passed back to the client application.	processMissingPropertyAnswer(MissingPropertyInfo mpi, String actionURI, boolean bRemember) will be detected as an action rather than the object of a new Statement providing the missing Property allowing client applications to not need to distinguish between answers and actions. In this case the additional arguments to the method will be null.
MappingAction	1. this instance of AcuityController 2. the data to be mapped 3. a Map containing the properties of the Individual mapping function	The static method is called to map the data for a graph series or a table column from the raw data to the data for presentation. Subclasses of the MappingAction class you differentiate between column ordered mappings and row ordered mappings, etc.

Types of AcuityControllerAction and Details of Usage

The table below lists the action-related Properties defined in the *apvf* ontology. The namespace prefix is omitted in the table.

Property Name	**Summary of Usage**
askUserProperties	When an ObjectProperty is made a subProperty of this Property, discovery of a missing Property of the subProperty type will result in a "suggestion" to the client that the user should be asked for the object (Individual of range Class) to be used in a new Statement with predicate matching the subProperty and object the user-specified Individual.
askUserPropertiesIncludeCreateNew	When an ObjectProperty is made a subProperty of this Property, any list of Individuals offered to the user as options for a missing Property will include the "action" option of creating a new Individual to be the object of a new Statement matching the missing Property, i.e., a new

	Individual with type being the Class which is the range of the subProperty. (Note: this Property is a subProperty of askUserProperties above.)
hasAskUserRDQLValueFilter	When a Class has a hasValue Restriction on this Property, the String value is used as an RDQL query string to generate the list of Individuals which can be objects in the missing Statement.
autoInstantiateProperties	When an ObjectProperty is made a subProperty of this Property, discovery of a missing Property of the subProperty type will result in automatic creation of a new Individual of the range Class which will be used as the object of a new Statement with predicate matching the subProperty and object the new Individual.
hasAction	The range of this ObjectProperty is the Class AcuityControllerAction, which has DatatypeProperties identifying the package, class, and static method name of a Java procedure to be executed when the appropriate conditions, determined by subProperty distinctions, are met. The subProperties of this Property are the Properties actually instantiated in an application ontology, i.e. instances of hasInstanceCreationAction, hasMissingObjectPropertyAction, and hasOnDemandAction.
hasInstanceCreationAction	(subProperty of hasAction) When a Class has a hasValue Restriction on this Property, the Individual of type InstanceCreationAction which is the value defines the action to be taken as a side effect when a new Individual of the Class is created.
hasMissingObjectPropertyAction	(subProperty of hasAction) When a Class has a hasValue Restriction on this Property, the Individual of type

	MissingObjectPropertyAction which is the value defines the action to be taken when the Class is in the range of the missing Property.
hasOnDemandAction	(subProperty of hasAction) When a Class has a hasValue Restriction on this Property, the Individual of type OnDemandAction which is the value defines an action which is available as a user-invoked action.
includeAsMissingPropertyOption	When an Individual of type OnDemandAction includes a hasValue Restriction on this Property with value true, the Individual is included in the List of possible answers given to the client for an "askable" missing Property.

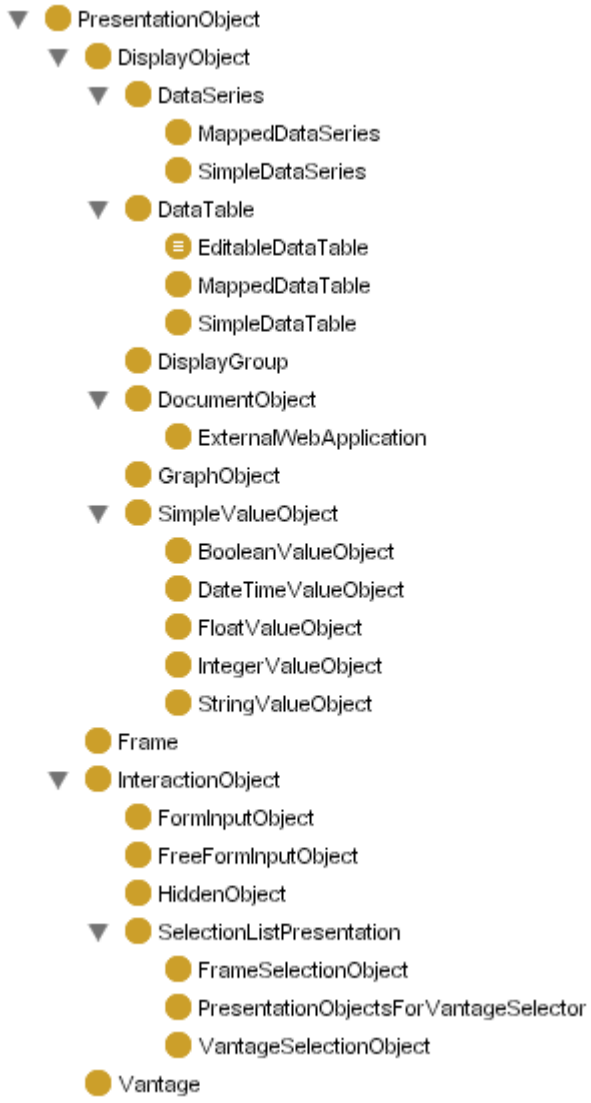
Action-Related Properties of the Acuity Problem/Vantage/Frame Ontology

A.8 Implementation of the Problem/Vantage/Frame Paradigm

A.8.1 PresentationObjects

Presentation Objects (POs) populate a Vantage and stand between the raw data and the presentation. It is useful to distinguish between POs used to display information to the user, such as a graph of one or more data series, and POs used to obtain information from the user and pass it to the AcuityController. This gives rise to two subclasses of PresentationObject: 1) DisplayObject (DO) and 2) InteractionObject (IaO). In the former case, a particular Vantage type will have certain DOs which will be associated with certain data sources (WorkDomainInformationObjects or WDIOs). A flexible implementation might allow a user to add additional DOs with their associated WDIOs to a Vantage. In the latter case, an IaO maps user input to possible assertions and/or deletions of RDF triples or to actions (see discussion of actions above). An IaO can be created from a InteractionContent Java class instance, which defines the data content of the IaO. Independent of any missing properties, IaOs may also be associated with a Vantage enabling the user to assert some Statement or set of Statements, such as [instanceOf AcuityController, hasFrame, instanceOf Frame], at any time.

The apvf:Frame and apvf:Vantage classes themselves are also types of apvf:PresentationObject. The ObjectProperty apvf:containsPresentationObject has apvf:PresentationObject as both its domain and range and is inverse functional. In other words, every PO can contain any number of POs, but a PO can only be contained by (inverse Property apvf:isContainedByPresentationObject) a single PO. A partial hierarchy of InformationObjects in the Problem/Vantage/Frame ontology, including apvf:PresentationObject and some of its subclasses, are shown in the figure below.



The mapping between WDIOs and DOs captured by the ObjectProperty apvf:isEncodedBy, which has domain apvf:WorkDominInformationObject and range apvf:InformationObject. Its inverse Property is apvf:encodes. For example, the class apvf:CachedRowsetDataTable has a hasSomeValuesFrom restriction on the apvf:encodes Property with range apvf:LegacyRelDataSet or apvf:StoredProcedure. In other words, an rowset returned from a relational database by an SQL query or a stored procedure can be encoded and therefore displayed in an instance of an apvf:CachedRowsetDataTable. As a more complex example, an apvf:GraphObject contains one or more apvf:DataSeries, which in turn encode an apvf:LegacyRelDataSet or an apvf:StoredProcedure.

A.8.2 Presentation Parameters

Presentation Parameters are those characteristics of PresentationObjects used by the user-interface to achieve an actual display. Presentation Parameters include things like Position (xPos, yPos, zOrder, etc.) and DisplayType (ListBox, DropdownListBox, etc.). Some Presentation Parameters are associated with a PresentationObject by

DatatypeProperties, e.g., apvf:allowsMultipleSelection with range xsd:boolean. Other Presentation Parameters are "composite" in that the values are themselves Individuals of type Parameter associated by an ObjectProperty with the PresentationObject and with "indirect" DatatypeProperty values. The owl:Class Position is an example of a composite Parameter. The x and y coordinates (xPos, yPos), the zOrder, the height, width, and depth of a PresentationObject are aggregated together into an instance of Position. Thus for a given PresentationObject "myPO", the following Statements might exist (in pseudo-syntax):

<#myPO>, <apvf:hasPosition>, <#myPOsPosition>
<#myPOsPosition>, <apvf:xPos>, 321
<#myPOsPosition>, <apvf:yPos>, 452
<#myPOsPosition>,<apvf:zOrder>,0

There are two reasons why values of DatatypeProperties might be aggregated together to form composite or indirect parameters. The first is for convenience of organization, the reason for creating the <apvf:hasPosition> ObjectProperty with range Position. The second, and more compelling, is in order for PresentationObjects to persistently share parameters. The value of a DatatypeProperty is not a uniquely identified element of an OWL ontology and therefore cannot be shared between PresentationObjects. Rather the value, e.g., true for the apvf:allowsMultipleSelection, would be saved with each instance of the modified PresentationObject and there would be no way to represent (and remember) that the two POs have the value in common. A good example of this is the class apvf:Color, the range of the apvf:hasColor Property. It is quite conceivable that in some kind of brushing operation (highlighting the same data in multiple views), two POs might want to use the same Color. The RDF statements below would result:

<#myPO1>, <apvf:hasColor>, <#aColor>
<#myPO2>, <apvf:hasColor>, <#aColor>
<#aColor>,<blueVal>,<0>
<#aColor>,<greenVal>,<0>
<#aColor>,<redVal>,<255>

Now <#aColor> might be changed by changing it's DatatypeProperties, but the two POs will continue to share that same Color, whatever it becomes. For more information about when a composite is changed directly and when a copy is made and then changed, see the discussion of Default Values below.

Note that while some parameters are single-valued (functional), as were the examples above, others can be multi-valued. A good example is the HighlightRegion class. A Graph2D type DisplayObject instance might have multiple instances of the composite Parameter HighlightRegion. Thus the apvf:hasHighlightRegion can have multiple values (objects) for a given PresentationObject (subject). The composite HighlightRegion has Parameter apvf:hasRegion with range Region. Region in turn has Parameters apvf:hasColor with range Color, and apvf:hasSlice with range Slice and/or hasDiscreteValue with range xsd:string. Slice in turn has as its Parameters the xsd:float

values loThreshold and hiThreshold. This suggests Statements similar to the following (in pseudo-syntax):

<#my2DGraph>, <apvf:hasHighlightRegion>, <#hlr-1>
  <#hlr-1>, <apvf:hasRegion>, <#rgn-1>
    <#rgn-1, <apvf:hasSlice>, <#slice-1>
      <#slice-1>, <apvf:loThreshold>, 0
      <#slice-1>, <apvf:hiThreshold>,100
    <#rgn-1>, <apvf:hasColor>, <apvf:blue>
<#my2DGraph>, <apvf:hasHighlightRegion>, <#hlr-2>
  <#hlr-2>, <apvf:hasRegion>, <#rgn-2>
    <#rgn-2>, <apvf:hasSlice>, <#slice-2>
      <#slice-2>, <apvf:loThreshold>, 200
      <#slice-2>, <apvf:hiThreshold>,300
    <#rgn-2>, <apvf:hasColor>, <apvf:red>
<#my2DGraph>, <apvf:hasHighlightRegion>, <#hlr-3>
  <#hlr-3>, <apvf:hasRegion>, <#rgn-3>
    <#rgn-3>, <apvf:hasSlice>, <#slice-3>
      <#slice-3>, <apvf:loThreshold>, 500
      <#slice-3>, <apvf:hiThreshold>,600
    <#rgn-3>, <apvf:hasColor>, <apvf:yellow>

For more details on using SharedCompositeParameterSets to accomplish effects such as data brushing, e.g., the sharing of highlight regions across presentation objects, see SharedCompositeParameterSets below.

A.8.1.2 PresentationParameterMap

The PresentationParameterMap Java class is a helper to the AcuityController class. For a given PresentationObject, a call to getPresentationObjectParameters returns an instance of the class PresentationParameterMap. Each PresentationParameterMap contains a subject (Individual of type PresentationObject or Parameter) and a Hashtable containing key/value pairs. (Actually, PresentationParameterMap extends java.util.Hashtable so the Hashtable behavior is entirely inherited.) Each key is a predicate (OntProperty) with the namespace removed and each corresponding value is an instanceOf one of the following:

- a Literal - a single DatatypeProperty value; the type and the value retrieved from the Literal with getDatatype( ), getInt( ), getBoolean( ), etc.
- a Literal[] - an array of values for a multi-valued simple parameter
- an Individual - the value is an Individual such as apvf:ListBox, which is an Individual of type DisplayType and a possible value for apvf:hasPresentationNature
- an Individual[] - an array of Individuals; possible when a multi-valued parameter can have Individuals as value

- a PresentationParameterMap - for composite parameters; the subject is the object (value) of the containing PresentationParmeterMap
- a PresentationParameterMap[] - for multi-valued composite parameters; the subject of each element is the object (value) of a PresentationParmeterMap at the higher level--see example below

Relating this to the example of multiple HighlightRegions above, the PresentationParameterMap returned by getPresentationObjectParameters("#my2DGraph") has subject "#my2DGraph" and a key "hasHighlightRegion" (and perhaps others). The value of this key is a PresentationParameterMap[] of size 3 whose elements have subjects "#hlr-1", "#hlr-2", and "#hlr-3", respectively. Each of the three PresentationParameterMaps will have the two keys "xMin" and "xMax" with the associated Literal values (0,100), (200,300), and (500,600).

The AcuityController provides a convenience method to get Statements reflected in a PresentationParameterMap:

StmtIterator getMatchingParameters(PresentationParameterMap ppm, String propLocalName)

The StmtIterator allows access to the set of Statements which are associated with the first match of a key with propLocalName in the map or a sub-map. Note that only Statements associated with the first match is returned. If multiple matches exist in the map or its sub-maps, they will not all be returned.

The PresentationParameterMap class itself also has a convenience method for getting information from maps/submaps. Unlike the AcuityController's getMatchingParameters(..), this method does not query the ontology but returns just what is currently found in the map. This method's signature is:

Hashtable getMatchingParameters(String propLocalName)

The returned Hashtable has subjects as keys and non-map objects as values for all entries matching the input predicate and not having an object which is a map--in other words, the map is "flattened."

A.8.2.2 Setting Parameters

The AcuityController provides a convenience method for setting/updating PresentationParameterMaps and the underlying ontology simultaneously:

boolean updatePresentationParameterMap(PresentationParameterMap ppm, String propLocalName, Object newVal)

The PresentationParameterMap (ppm) is normally retrieved from an InteractionObject. The newVal argument can be a String which names an Individual, a String which may be

converted to a number or a boolean value, or an Integer or Boolean. In any case, the newVal will be converted either to an Individual for an ObjectProperty or to a Literal for a DatatypeProperty. If the propLocalName identifies a DatatypeProperty, the rdfs:range will be retrieved and newVal will be validated. Invalid types will cause an AcuityException to be thrown.

A.8.3 Shared Composite Parameter Sets

As discussed above, a composite Parameter is an Individual of type Parameter which is the object of a "hasPresentationParameterObject" Property on a PresentationObject or a higher-level composite Parameter. In some circumstances, e.g., to accomplish "brushing" of data (highlighting the same or related data in the same color for example) across two or more tables or graphs, it is desirable to have some way of identifying a set of Individuals that share a common composite Parameter, e.g., a highlight region. The mechanism for doing this is the SharedCompositeParameterSet.

In the case of brushing between tables or graphs, the HighlightRegion has the composite Parameter Region (through ObjectProperty hasRegion), which has the composite Parameter Slice with Parameters loThreshold, hiThreshold. HighlightRegions on two or more tables or graphs can be placed, either at ontology design time or at runtime, in a SharedCompositeParameterSet, meaning that the HighlightRegions are the objects of a "hasMember" ObjectProperty of the SharedCompositeParameterSet. The SharedCompositeParameterSet ObjectProperty hasSharedParameter will have as object an Individual of type Region. The net result will be that each HighlightRegion in the set will have the same Region Parameter. Editing this Parameter on one of the HghlightRegions will edit it for all (since they all share the same one in the ontology).

A SharedCompositeParameterSet is a subclass of the more general Set class. By calling the method getSetMembership(String setUri) or getSetMemberships(String memberUri), an AcuityController client can always get the PresentationObjects or higher-level Parameters that share a composite Parameter. This may be useful, for example, in updating the display if the shared Parameter is edited.

A.8.4 AcuityController Methods Supporting SharedCompositeParameterSet

A SharedCompositeParameterSet is created by calling the AcuityController method:

> Individual createSharedCompositeParameterSet(String setLocalName, String sharedParameterPropertyName, List members)

The first argument, setLocalName, can be null in which case a name will be generated. The last argument is a List of member Individuals or their URIs. The shared Parameter of the first member in the List becomes the Parameter shared by all members of the set. The method returns the Individual which is the new SharedCompositeParameterSet.

Members can be added to an existing SharedCompositeParameterSet by calling:

boolean addSharedCompositeParameterSetMember(Individual set, Individual newMember)

or the equivalent method taking the Set and new member's URI strings as arguments.

To remove a member from a SharedCompositeParameterSet, creating an independent version of the composite Parameter for the removed member of the set, call:

void removeSharedCompositeParameterSetMember(Individual set, Individual removalMember)

Note: when the last member is removed from a set, the set is dissolved.

To destroy a SharedCompositeParameterSet, after first creating an independent version of the composite Parameter from each member of the set, call:

void dissolveSharedCompositeParameterSet(Individual set)

A.8.5 InteractionObjects, Extended RDQL (xRDQL), and Anchors

Since an IaO provides a mechanism for the user to modify the knowledge base of the AcuityController, it can be thought of as providing access to a set of insertions, deletions, or updates of RDF triples in the ontology's instances namespace (aBox). Unfortunately, RDQL has not been extended to support INSERT, DELETE, and UPDATE, although the need has been recognized. The AcuityController implementation must assume such an RDQL extension, and uses the following syntax:

- INSERT (<subject>, <predicate>, <object>)[, (<subject2>, <predicate2>, <object2>)[,...]]
- DELETE (<subject>, <predicate>, <object>)[, (<subject2>, <predicate2>, <object2>)[,...]]
- UPDATE (<subject>, <predicate>, <object>)[, (<subject2>, <predicate2>, <object2>)[,...]]

where the square brackets indicate optional additional triples.

Some assumptions are required to implement these additional operations.

1. An INSERT will generally add new triples. However, to make use easier (require less knowledge on the part of the author of a statement) an INSERT of a triple which would result in a cardinality violation will have the effect, where possible, of doing an UPDATE.
2. An UPDATE will only succeed where there is no triple or there is a single triple already existing that matches the UPDATE pattern. The first case is supported to require less knowledge on the part of the statement author. In the case of an existing triple, that triple will be removed and a new triple added with the new

information. If there are multiple triples that match the pattern, which one to update is ambiguous and an AcuityException will be thrown.

3. A DELETE maps to the Jena removeAll(Resource s, Property p, RDFNode o), which removes all matching Statements and which allows one or more of the arguments to be null to indicate that any value of that argument will match. Rather than "null", an unbound variable, e.g., ?x, may be placed as the unconstrained argument.

4. Any subject, predicate, or object in a triple of an INSERT, DELETE, or UPDATE may contain an RDQL SELECT statement. Since the SELECT itself may have any number of comma-separated triples, it is wrapped in square brackets to make its extent easier to parse. If the RDQL SELECT returns multiple values, the INSERT, DELETE, or UPDATE will be done only for the value which the user identifies from the set (implying an interaction with a menu of choices).

Note that the SELECT used for the subject, predicate, or object of the triple, as described in #4 above and which is enclosed in square brackets, can be a conjunction (union) and/or a difference of the form:

[SELECT ... PLUS SELECT ... MINUS SELECT ...]

This is supported because the limitations of the RDQL SELECT make it otherwise difficult to combine and/or reduce a set of possible answers, e.g., provide a list of possible answers which are all of the POs in a Frame except those POs in the current Vantage.

In addition to adding INSERT, DELETE, and UPDATE to our extended RDQL (xRDQL), we need, in order to implement IaOs, someway of "anchoring" at design time the value of a subject or object at runtime. The ontology as designed identifies primarily the Classes of things that exist in the problem/vantage/frame implementation or in the work domain. It is usually not until the AcuityController is used that Individuals belonging to these Classes are created, and it is these runtime Individuals which serve as subjects and objects of RDF triples in the knowledge base. To facilitate this anchoring, the owl:Class apvf:InteractionAnchor is created. It has Individuals "self", "currentAC", "currentFrame", "currentUser", "focusVantage", and "containingPO". (The anchors "useRange" and "useAnswer" are special cases used in the apvf:hasEffect of IaO's as described below.) Note that the syntax is to place ${...} around the anchor in the query string.

To relate xRDQL expressions to a particular subclass of IaO, the DatatypeProperty apvf:hasEffect is implemented with domain apvf:InteractionObject and range xsd:String. An IaO may have multiple hasEffect properties, but the order of execution is not guaranteed. Therefore, any statements which are order-dependent should be placed in a single hasEffect value. The syntax of the value is simply a series of semi-colon-separated INSERT, DELETE, or UPDATE statements.

The following examples are meant to clarify the use of xRDQL and IaOs and to illustrate the usage of members of the InteractionAnchor Class as subjects and objects.

1. An apvf:FrameSelectionObject, subclass of apvf:SelectionListPresentation which is a subclass of apvf:InteractionObject, is meant to allow the user to switch to another apvf:Frame or create a new one at any time. The Class is therefore given the necessary restriction that the Property hasEffect has value:

   UPDATE (${currentAC}, <apvf:hasFrame>, ${useRange})

   At runtime, the subject "${currentAC}" is replaced with the Individual that is the ontology instance of the current AcuityController. The object is the user's selection from the set of possibilities. (The "${userRange}" is initially processed to determine the set of possible answers and then replaced with "${useAnswer}" when the client returns a value.) This set is generated exactly as described above for missing properties: the eligible frames are defined by the apvf:hasAskUserRDQLValueFilter DatatypeProperty on Frame, if there is one, and the option of creating a new Frame is included because "hasFrame" is an rdfs:subPropertyOf askUserPropertiesIncludeCreateNew.

2. An aePresentation:FleetSelectionObject is meant to allow the user to select an apvf:Fleet instance and make it the current Fleet of the current Frame. The aePresentation:FleetSelectionObject Class has the Property apvf:hasEffect with value:

   UPDATE (${currentFrame}, <aePresentation:hasCurrentFleet>, ${useRange})

   In this case the Fleet Class does not have any special Properties but since "hasCurrentFleet" has the Class Fleet as range, all instances of Fleet are in the set from which the user selects. The user's selection is the object of the updated RDF triple.

3. The previous example is reexamined without the use of "currentFrame" and "useRange" to illustrate the use of RDQL SELECT statements. The value of the hasEffect could be:

   UPDATE ([SELECT ?frame WHERE (${currentAC}, <apvf:hasFrame>, ?frame)], <aePresentation:hasCurrentFleet>, [SELECT ?fleet WHERE (?fleet, <rdfs:type>, <apvf:Fleet>)])

4. The aerf:HighlightEngineSelection allows the user to select a different engine with the desired side-effect of changing to the EngineDetail Vantage with the new Engine. The hasEffect value is:

   UPDATE (self, <aerf:hasEngine>, [SELECT ?engine WHERE (${focusVantage}, <aerf:hasFleet>, ?fleet), (?fleet, <apvf:hasMember>, ?engine)]);

UPDATE (${currentFrame}, <apvf:hasFocusVantage>, [SELECT ?vantage WHERE (?vantage, <rdfs:type>, <aerf:EngineDetail>), (${currentFrame}, <apvf:hasVantage>, ?vantage)])

The first UPDATE allows the user to choose an Engine from those which are members of the current Fleet and associate it with the current ??. The second UPDATE changes the focus Vantage. Note that when a statement has a single object, as the second is expected to, the UPDATE is made automatically without user interaction.

The pre-defined anchors which can be used in an extended RDQL statement (Individuals of Class apvf:InteractionAnchor) are summarized in the following table:

Anchor Identifier	**Will be Replaced With**
self	The URI of the Individual which is the current InteractionObject
currentAC	The URI of the Individual corresponding to the current AcuityController
currentFrame	The URI of the Individual which is the current Frame associated ("hasFrame") with the current AcuityController
currentUser	The URI of the Individual which corresponds to the current user
focusVantage	The URI of the Individual Vantage which has the focus
containingPO	The URI of the Individual PresentationObject which contains the current InteractionObject
useRange	"useAnswer" after the set of possibilities is set from the Class which is the range of the Property having this object.
useAnswer	The URI of the Individual which has been designated in the interaction as the answer
now	The current date/time

The concept of anchors is made extensible through the Class apvf:InteractionAnchor, which must have DatatypePropery rdqlQueryString for non-built-in anchors. The results

of the anchor query are used in the referencing query. For example, an instance of apvf:RDQLAnchor with name "aerf:currentFleet" can be created with associated query:

> SELECT ?fleet WHERE (${currentFrame}, <aerf:hasCurrentFleet>, ?fleet)

Now a query to get all of the Engines in the currentFleet could be written:

> SELECT ?engine WHERE (${currentFleet}, <apvf:hasMember>, ?engine)

Note that anchors can also be used in the SQL statements which are the values of the apvf:dbSQLString DatatypeProperty of the apvf:LegacyRelDataSet Class and in the apvf:storedProcedureSignature DatatypeProperty of the apvf:StoredProcedure Class. This allows SQL queries and stored procedure arguments to use specified runtime substitutions. For example, using the definition of the anchor "currentFleet" given above, an SQL query might be written as:

> SELECT ESN FROM Engines WHERE Fleet = '${currentFleet}'

The results of an anchor's RDQL query may be referenced by name, as would be necessary if the query returned multiple variables. For example, if the currentFrame had a "hasCurrentEngine" as well as hasCurrentFleet, the anchor "currentFleetAndEngine" might be defined as:

> SELECT ?fleet, ?engine WHERE (${currentFrame}, <aerf:hasCurrentFleet>, ?fleet), (${currentFrame}, <aerf:hasCurrentEngine>, ?engine)

Then an SQL query using this anchor to get the engine's hours and the owning Fleet's contact might look like:

> SELECT hours, OwnerContact FROM Engines, FleetContacts WHERE ESN = '${currentFleetAndEngine:engine}' AND Fleet = '${currentFleetAndEngine:fleet}'

A.8.6 InteractionObjects and Missing Properties

An InteractionObject instance can be created because a particular Vantage (or other containing PO) has an IaO associated with it at the abstract Class level or because an askable missing Property is associated with the InteractionObject. This latter association is a bit problematic in the sense that it is perhaps most logical to associate the IaO with the Property which is missing and askable. However, using a Property as the Subject or Object of an RDF Statement would make the ontology OWL Full rather than the desired OWL DL. Therefore, we associate the IaO with the Class which is the range of the missing Property using a Restriction on the apvf:hasInteractionObject ObjectProperty.

A.9 Default Values, Specified and Learned

OWL does not currently support default values. Default values introduce non-monotonic reasoning and can significantly increase an application's complexity. However, there are times when the availability of default values for initializing a new Individual as the object

of an ObjectProperty or Literal as object of a DatatypeProperty is very useful. One way to provide default values is through the use of InstanceCreationActions, as described above. This approach requires writing some Java code, the action, as well as adding the action to the ontology. The AcuityController provides another way of implementing default values that involves only additions to the ontology and supports learned default values.

It is desirable to be able to set a default value for a given Property on new Individuals of a given class. Associating this information with the Property is not possible in OWL DL as Properties cannot be reified in the ontology so that they can have their own Properties. Furthermore, this approach would make the default value information visible to other OWL reasoners and since there is currently no standard, might affect their behavior in undesirable ways. Even if making the ontology OWL Full were acceptable, the current tools ontology editing, e.g., the Protege OWL plugin, are geared toward OWL DL ontology development.

Classes (and Properties) in OWL can have one of several "annotation properties" including rdfs:comment and rdfs:label. Another annotation is called rdfs:seeAlso, which "is used to indicate a resource that might provide additional information about the subject resource" ("RDF Vocabulary Description Language 1.0: RDF Schema," http://www.w3.org/TR/rdf-schema/). In this instance, we wish to do exactly that--provide additional information that will be used by the AcuityController. We create an apvf:Refining subclass called apvf:DefaultValue with subclasses apvf:ObjectDefault, apvf:IntegerDefault, apvf:BooleanDefault, etc. The subclasses (apvf:ObjectDefault, apvf:IntegerDefault, apvf:BooleanDefault, etc.) have apvf:hasObjectDefault (ObjectProperty), apvf:hasIntDefault (DatatypeProperty), apvf:hasBooleanDefault (DatatypeProperty), etc., respectively, with range Individual, xsd:int, xsd:boolean, etc. These associations allow the specification of the value of the default.

The simplest use of an apvf:DefaultValue instance is as the object of an rdfs:seeAlso annotation on a Property. This usage implies that this is always the default value when the Property occurs.

A more fine-grained use restricts application of the default to triples whose subject comes from a specified class. The apvf:DefaultValue class is the domain of the apvf:appliesToPropertyWithName DatatypeProperty (range xsd:string), which will have as value the name of the Property to receive the default value. With this set of Classes and Properties, we can now add an rdfs:seeAlso annotation to any Class and specify the default for a given Property. For ObjectProperties, the value of the annotation will be an Individual of type apvf:ObjectDefault. For DatatypeProperties, the value of the annotation will be an Individual of the class corresponding to the Property's range, e.g., IntegerDefault. In either case, the apvf:appliesToPropertyWithName value will be the name of the Property to receive the default value and the other Property (e.g., hasIntDefault) will provide the actual default value.

As an example, we wish the apvf:InteractionObject subclass apvf:FrameSelectionObject to have as default value of apvf:hasPresentationNature the Individual "ListBox", an instance of the class apvf:DisplayType. We create an rdfs:seeAlso annotation with object

an instance of ObjectDefault with apvf:appliesToPropertyWithName value "apvf:hasPresentationNature" and apvf:hasObjectDefault value apvf:ListBox.

Note that one undesirable characteristic of this approach is that the apvf:appliesToPropertyWithName value is not a direct link to the actual Property to receive the default value but only an xsd:string which must match the Property's name. Consequently, if the name of the Property is changed, this xsd:string value must be edited as well to keep it synchronized with the actual Property name. Of course when it is sufficient to associate a default value with the Property alone--the default will be the same regardless of the class to which it is applied--the rdfs:seeAlso annotation is applied to the Property instead of the Class and apvf:appliesToPropertyWithName is not used.

As an example of an rdfs:seeAlso on a Property, the apvf:hasRowsetMaxRows Property has domain apvf:CachedRowsetDataTable and specifies the maximum number of rows to request from a legacy relational database table (so that if the actual number of rows is enormous the application will not stop responding). By creating an rdfs:seeAlso annotation on apvf:hasRowsetMaxRows with value an instance of apvf:IntegerDefault with apvf:hasIntDefault value 150, the AcuityController is provided with a default value that can be used to initialize the apvf:hasRowsetMaxRows Property on any Individual that it creates.

All instances of apvf:DefaultValue may have the additional DatatypeProperties apvf:thresholdFrequency and/or apvf:minimumSampleSize. The first, if present, indicates that the AcuityController should look at the historical instance data to see if some value has occurred frequently enough to be used as the default. If so, this learned default will be used instead of the specified default. The threshold frequency is expressed as a decimal fraction, e.g., 0.60 meaning this value occurred at least 60% of the time. The second Property, if present, indicates a minimum number of observations necessary before the learned default will be used. For example, if apvf:minimumSampleSize is 20, then at least 20 historical instances must be in the histogram before the learned default, assuming the threshold is met, will be used. If apvf:minimumSampleSize is specified but apvf:thresholdFrequency is not specified, as soon as the sample size requirement is met the most common historical value will be used as the default regardless of frequency. In the case of ties, the choice among the top contenders will be arbitrary.

More specifically, the histogram of values from which a learned default value of a given Property ("thisProperty") for a given Individual ("thisInstance") translates into the following RDQL query:

> SELECT ?x WHERE (<URI of thisInstance>, <rdf:type>, ?instType), (?otherInst, <rdf:type>, ?instType), (?otherInst, <URI of thisProperty>, ?x)

This returns all of the values which "thisProperty" has been given for all Individuals of the same rdf:type as the current subject Individual. These values can be Individuals (ObjectProperties) or Literals (DatatypeProperties).

The default value created by the RDQL query above is learned from all Individuals of the same rdf:type regardless of who (which user) generated the instances. Obviously this

approach cannot learn user preferences. To handle user preferences we must restrict the learned value to those Individuals of the same rdf:type that were created by the current user. A more general user-based learned value could, say in the absence of sufficient data for a [new] user, look at preferences of users of the same rdf:type, e.g., other planners. Some thought is needed as to the meaning of "users of the same rdf:type." Certainly, all users have the same type at the owl:Thing level. Our approach is to expand the user restriction to all users who share some Property or Properties with the current user, e.g., hasSkill.

The RDQL queries for these two situations (this user and the set of users sharing the Property apvf:hasSkill with this user) are, respectively:

> SELECT ?x WHERE (<URI of thisInstance>, <rdf:type>, ?instType), (?otherInst, <rdf:type>, ?instType), (?otherInst, <apvr:hasUser>, currentUser), (?otherInst, <URI of thisProperty>, ?x)
>
> SELECT ?x WHERE (<URI of thisInstance>, <rdf:type>, ?instType), (?otherInst, <rdf:type>, ?instType), (?otherInst, <apvr:hasUser>, ?u), (?u, <apvf:hasSkill>, ?skill), (currentUser, <apvf:hasSkill>, ?skill), (?otherInst, <URI of thisProperty>, ?x)

This gives rise to two more optional DatatypeProperties on instances of apvf:DefaultValue: 1) apvf:userPreference with range xsd:boolean, and 2) apvf:userGroupDefinedBy with range xsd:string. A value of true for apvf:userPreference causes the search for a learned value to begin with the target Property for Individuals of the same owl:Class with the same value of apvf:hasUser as the current subject.

If the threshold or sample size conditions for an apvf:DefaultValue with apvf:userPreference true are not met, one of two things will happen. If there are no apvf:userGroupDefinedBy Properties on the default, the specified default will be used. However, if one or more values of apvf:userGroupDefinedBy are present, the values are used to identify a set of users with the same values of the specified group Properties and a learned default is sought from the Individuals of the subject type and owned by this group of users. This causes the set of users to be expanded from the current user to all users who share the named Property or Properties with the current user. Because apvf:userGroupDefinedBy can be multi-valued (is not functional), the user group from which the value is learned can be restricted by more than one shared Property. If this set does not meet the threshold or sample size conditions, the specified default value is used.

As a final degree of flexibility in setting default values, the RDQL query used to identify the members of the histogram can be explicitly given as the value of the DatatypeProperty learningSetQuery (range xsd:string). The query should have a single selected variable, like the ?x in the three examples above, and should return a set of Individuals or Literals.

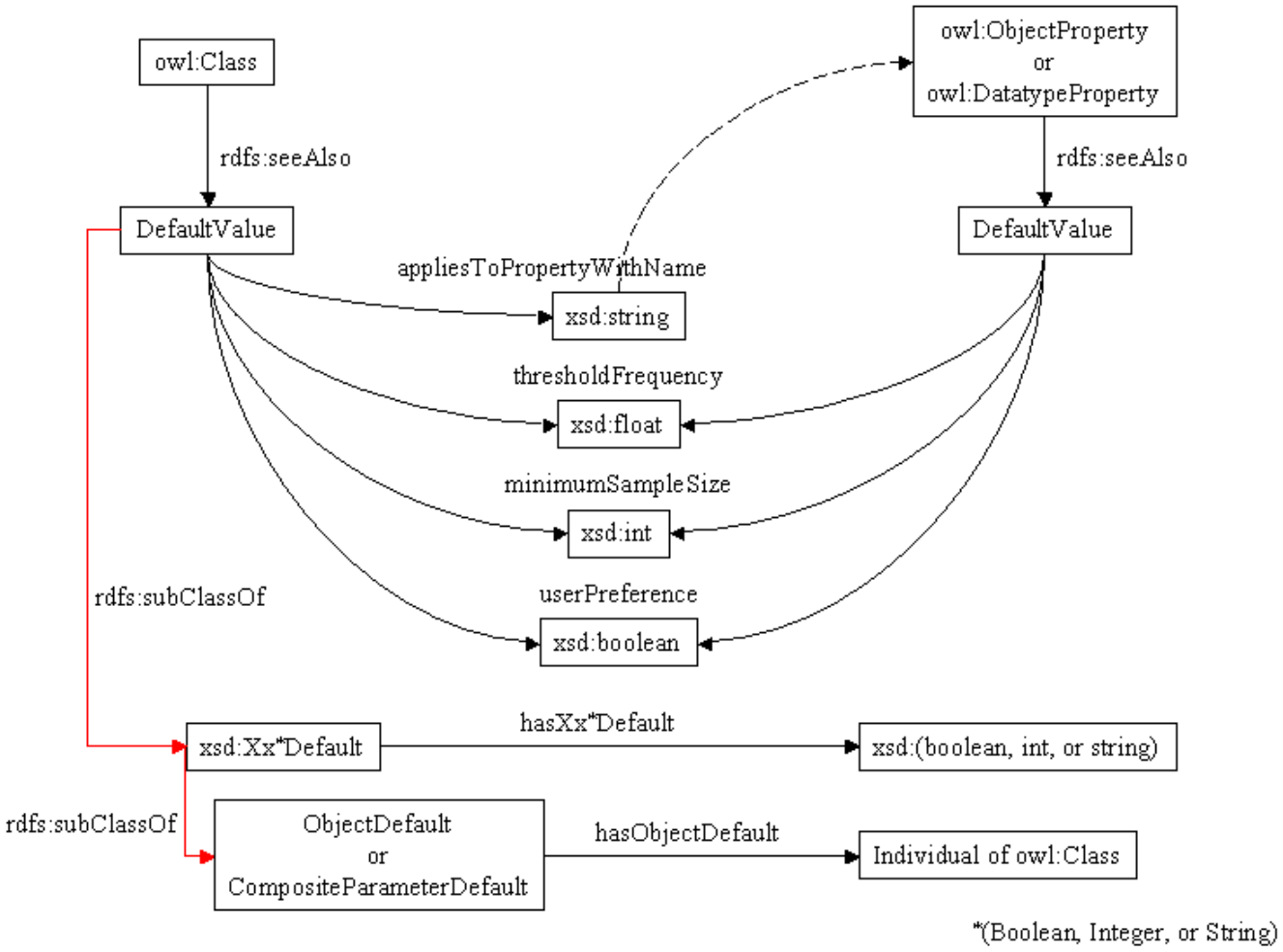
Note that group-based learning has not yet been implemented.

Learned defaults, particularly user preferences, are useful for editable Parameters. However, some Parameters, like apvf:Position and apvf:Color, are composite Parameters meaning that the Parameter value is an Individual which in turn has sub Parameters which are Literals. To learn a default value for a composite Parameter, we do not necessarily want to compare the various Individuals (the instance of Color or Position), but rather look at their sub Parameters.

Namespaces are used to determine when a complex Parameter can be re-used and when it must be duplicated. It is important that DefaultValues be defined in a namespace other than that of the instance data. When a complex Parameter has a value, i.e., a default value, which is an Individual from a namespace different from the instance data, and a call is made to the AcuityController to update that value in the PresentationParameterMap, the value is cloned, the clone is substituted for the original in the ontology and the map, and the changes are applied to the clone.

There is a related behavior relative to learned values of complex Parameters. If the dominant value from the instance data is a value from another namespace, e.g., a default value, then the new subject for which a value is sought will be given this dominant instance. However, if the dominant instance is from the instance data namespace, it is always cloned so that it may be independently edited.

The figure below summarizes the concepts and relationships used to implement default values.



Concepts and Relationships Implementing Default Values and Learned Default Values

Appendix B. Useful Concepts from Visualization Science

In focusing on the unifying, work-centered aspects of the user-interface design, one must not loose sight of a large body of research that informs us of various ways to effectively display information. Proven concepts in this field range from visual and graphical methods of making complex relationships seem intuitive to the use of pre-attentive properties to make key pieces of information arrive in the user's consciousness without cognitive effort (Card et al., 1999; Healey et al., 1996). Principles of visual representation must be understood and vigorously applied to achieve the project objectives, for vision is the predominant mode by which an unimpaired human being perceives the world (Pinker, 1984; Wade & Swanston, 1991) and is obviously the primary mode of computer-to-human communication in today's screen-based, point-and-click computing environment.

Vision pioneer David Marr (1982) observed, "A representation is a formal system for making explicit certain entities or types of information, together with a specification of how the system does this." It follows that any particular representation will make explicit certain information at the expense of other information that is pushed into the background and made more difficult to recover. Understanding what makes information easier or more difficult to perceive is key to effective use of the display in a human-computer interface. Visual representation is often categorized as sentential (sequential) or diagrammatic (graphical) (Larkin & Simon, 1987; Crapo et al., 2002). Natural language is an example of a sequential representation. While some languages are written left to right, others top to bottom, etc., the fact remains that the information is primarily encoded in a linear (one-dimensional) manner. Such is the case with the text of this document.

By contrast, information in diagrams is encoded by position in a 2-D (or higher) substrate. Perception is not usually sequential, but rather the observer's focus will change by following implied relationships in the diagram. Figure 1 is an example of this type of representation. One's eyes might initially focus on the depiction of a computer and monitor at the top of the concentric circles. From there one might traverse the connection to the descriptive text captioned "Adaptive Human-Computer Interface" or one might traverse the double-ended arrows leading to "User Models" or "Work Domain Models." Other relationships in the diagram are less explicit, such as the relationship of the bulleted list to the previously mentioned elements of the diagram. As in this figure, representations often use a combination of sentential and diagrammatic elements. As another example, a table uses a 2-D grid to organize information by intersecting classification schemes, identified by the row and column headings. The contents of the table cells may be text (sentential) or may be themselves graphic elements. Windowing systems, such as Microsoft Windows, use rectangular display elements (windows) to contain information but do not generally provide an organized manner of arranging the windows within the display. The Focus-Periphery principle described in papers on the subject of WCSS suggests a graphical layout that places important, more detailed information in the center of the display area and less important information in the periphery and is an example of a way of diagrammatically organizing screen content (Eggleston & Whitaker, 2002). The Problem-Vantage-Frame principle also associated with WCSS is a higher-level modeling concept concerned with determining the content of the screen, i.e. what information is in the focal region and what is in the periphery for

different work domain tasks (ibid). The possible ways in which information may be represented in a 2-D or pseudo-3-D display is less complex than one might expect. Most graphical presentations can be characterized by their use of the spatial substrate (position in the 2-D display), marks of some type, and the graphical properties of the marks, e.g. color (Card et al., 1999). Cleveland and McGill empirically investigated the accuracy of various means of graphically displaying quantitative data and identified the following ordering, from most accurate to least accurate (Cleveland, 1985):

• Position along a common scale

• Position along identical, nonaligned scales

• Length

• Angle and slope

• Area

• Volume

• Color hue, color saturation, and density (amount of black)

Tufte (1997) suggested that the Challenger disaster might have been avoided had the available data for all previous launches been shown as a 2-D plot of the amount of O-ring damage versus temperature. Card et al. (1999) note that the superiority of Tufte's representation derives from his use of the most important graphical characteristic, position in the spatial substrate, to display the most important information, the amount of O-ring damage and the temperature. As a note on information composition, Tufte (1997) also observes the criticality, in that instance at least, of displaying all of the data. When the launches without O-ring damage are not included in the data, the important pattern of increasing damage with decreasing ambient temperature is much less obvious.

While the choice of characteristics for representing a particular piece of information may be limited, the overall visual effect can be more (or less) than the sum of the parts due to the neurological hierarchy of the human visual processing system. Higher-level "laws" of human perception have been studied in considerable detail and are known as Gestalt principles (Card et al., 1999; Ware, 2000). These principles can have a guiding influence both on information content and information display. A few are listed here as examples.

- Every stimulus pattern is seen so as to simplify the resulting structure.
- Objects near one another tend to be grouped together into a perceptual unit.
- Elements are perceived much more strongly as a visual whole if there is symmetry in the whole.
- Neighboring elements are perceived as being grouped together when connected by straight or smoothly curved lines. (This is more powerful than proximity, color, size, or shape.)

These and other visualization principles can provide important guidance in designing the content and presentation of information in the WCSS displays. The Focus-Periphery approach's use of the center of the display for the most important information and the periphery for information anticipated to be less critical is an example of the use of the most important display characteristic—position in the spatial substrate—to arrange information on the screen by expected importance (see Eggleston & Whitaker, 2002). Based on principle, this makes sense if anticipated relevance of information is the most important characteristic to be perceived. Even if this is the case, other questions can arise. If the display is thought of in polar coordinates with the origin at the center of the screen, and relevance is now inversely proportional to distance from the center of the display (radius), what about angular position? Which things are close to each other at a given radius, which things are on the right, which on the left, etc., can be used constructively or destructively to assist the user.

Many such examples of potential application of visualization concepts could be conceived. Without guidance from validated principles of perception, user-interface design remains an art informed by intuition alone, but with appropriate application of such principles the design can be evaluated and discussed concretely. Of course these principles only inform the design of information content and presentation. The targeted user community must validate any real user interface to ensure that the principles have been correctly and effectively applied.

References

1. AFRL/HESS, 2002 – Broad Agency Announcement for Work Centered Support Systems for Autonomic Logistics.
2. Brickley, Dan, R.V. Guha, Brian McBride (ed.), 2004. *RDF Vocabulary Description Language 1.0: RDF Schema*. Available at http://www.w3.org/TR/rdf-schema/.
3. Card, Stuart K., Jock D. Mackinlay, and Ben Schneiderman, 1999. Readings in Information Visualization: Using Vision to Think, Morgan Kaufmann Publishers, San Francisco.
4. Cleveland, William S., 1985. The Elements of Graphing Data, Wadsworth Advanced Books and Software, Monterey, CA.
5. Crapo, Andrew, Laurie Waisel, William Wallace, and Thomas Willemain, 2002. *Visualization and Modeling for Intelligent Systems*, Intelligent Systems: Techniques and Applications (C.T. Leondes, ed.), CRC Press, Boca Raton, FL.
6. Dean, Mike, and Guus Schreiber, editors (2004): *OWL Web Ontology Language Reference: W3C Recommendation 10 February 2004*. Available on-line at http://www.w3.org/TR/owl-ref/.
7. Eggleston, R.G., Young, M.J., and Whitaker, R.D. (2000) *Work-Centered Support System Technology: A New Interface Client Technology for the Battlespace Infosphere*. Proceedings of NEACON 2000, Dayton, OH, 10-12 October, 2000, pp. 499-506.
8. Eggleston, R., and Whitaker, R. (2002). *Work-Centered Support Systems Design: Using organizing frames to reduce work complexity*, Proceedings of the Human Factors and Ergonomics Society 46th Annual Meeting, Human Factors and Ergonomics Society, Santa Monica CA, pp. 265 - 269.
9. Healey, Christopher G., Kellogg S. Booth, and James T. Enns, 1996. *High-Speed Visual Estimation Using Preattentive Processing*, ACM Transactions on Computer-Human Interaction, 3:2, pp. 107-135.
10. Jena—A Semantic Web Framework for Java, Open Source project at http://jena.sourceforge.net.
11. Hackos, JoAnn T. and Janice C. Redish, 1998. User and Task Analysis for Interface Design, John Wiley & Sons, Inc., New York, NY.
12. Horridge, Matthew, Holger Knublauch, Alan Rector, Robert Stevens, and Chris Wroe, 2004. *A Practical Guide To Building OWL Ontologies Using The Prot´eg´e-OWL Plugin and CO-ODE Tools Edition 1.0*, the University of Manchester, http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf.
13. Larkin, Jill H. and Herber A. Simon, 1987. *Why a Diagram is (Sometimes) Worth Ten Thousand Words*, Cognitive Science, 11, pp. 65-99.
14. Marr, David, 1982. Vision: A Computational Investigation into the Human Representation and Processing of Visual Information, W. H. Freeman and Company, San Francisco.

15. Newell, Allen, 1990. Unified Theories of Cognition, Harvard University Press, Cambridge, MA.
16. Pinker, Steven (ed.), 1984. Visual Cognition, Elsevier Science Publishers B. V., Amsterdam, The Netherlands.
17. Rector, Alan L., 2003. *Modularisation of Domain Ontologies Implemented in Description Logics and related formalisms including OWL*. Knowledge Capture, (October 23-25, 2003), ACM, pp. 121-8. http://www.cs.man.ac.uk/~rector/papers/rector-modularisation-kcap-2003-distrib.pdf
18. Sowa, John F., 2000. Knowledge Representation: Logical, Philosophical, and Computational Foundations, Brooks/Cole, Pacific Grove, CA.
19. Todd, Peter, and Izak Benbasat, 1999. *Evaluating the Impact of DSS, Cognitive Effort, and Incentives on Strategy Selection*, Information Systems Research, 10:4 (December 1999), pp. 356-374.
20. Tufte, Edward. R., 1997. Visual Explanations, Graphics Press, Cheshire, CT.
21. Vicente, Kim J., 2000. *HCI in the Global Knowledge-Based Economy: Designing to Support Worker Adaptation*, ACM Transactions on Computer-Human Interaction, Vol. No. 2, pp 263-280.
22. Ware, Colin, 2000. Information Visualization: Perception for Design, Academic Press, San Diego, CA.
23. Winograd, Terry (ed.), 1996. Bringing Design to Software, ACM Press, New York, NY.