An Ontology-Based Architecture for Adaptive Work-Centered User Interface Technology[†]

Amy Aragones, Jeanette Bruno, Andrew Crapo, Marc Garbiras
General Electric Global Research
1 Research Circle
Niskayuna, NY 12309

Abstract

Adaptive Work-Centered User Interface Technology (ACUITy) is an ontology-based approach to modeling and implementing intelligent user interfaces built on top of Jena. Its potential benefits and underlying technologies are explored in the context of decision support systems.

1 Introduction

This paper introduces the Adaptive Work-Centered User Interface Technology (ACUITy) software architecture and describes its semantic underpinnings. ACUITy embodies two novel concepts: 1) an ontology modeling approach to characterize a work domain in terms of "work-centered" activities, including the interface and interactions between the decision support system and the user, and 2) use of these semantic models of the human-machine interface to provide adaptive interaction, both user directed and automated, in the work-centered characterization and presentation mechanisms of the user interface.

The practical benefit of embodying these concepts in the ACUITy architecture is that work-centered decision support applications can be developed much more rapidly and flexibly. The semantic modeling approach, built on top of Jena, formalizes and simplifies this process. The developer creates a functioning application by extending the domain-independent ontologies of work and human-computer interaction. The design is revised by user-directed customizations and through system learning of user preferences.

In this paper we provide an overview of the ACUITy architecture, focusing on key concepts in the ontology design and application. Please note that ACUITy is a work-in-progress. Our research has reached the point where it will benefit from wider discussion, feedback and, hopefully, development contributions. Over the next few months we will be transitioning ACUITy to Open Source, making public the results of three years of research and inviting participation from the larger community in addressing the many open questions to be answered. We are hopeful that this will prove beneficial to all.

2 The Context: Adaptive Work-Centered Support

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.

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

We extend Eggleston and Whitaker's approach by taking advantage of recent developments in semantic technology to achieve a new level of adaptive decision support. In the ACUITy architecture, we explicitly characterize WCSS and domain concepts and relationships in a hierarchy of ontologies, associated with upper level ontological constructs that enable adaptive reasoning and extensibility.

Work-centered support systems are built upon the premise that providing 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.

Maintenance planning and logistics, the focus of the first ACUITy applications, 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 during application design by allowing the user to adjust information layout and content in real time to suit the context in which work is being performed. ACUITy tracks changes made by users and uses this instance data to learn about and extend the models. The following sections describe the ACUITy architecture and its implementation.

3 The Approach: Semantic Modeling

ACUITy uses the Web Ontology Language (OWL) to represent semantic models of the user, the user-interface, the work domain, and the information content relevant to work in the domain. These models are captured in a hierarchy of ontologies ranging from domain-independent upper-level and WCSS ontologies to domain-specific ontologies of a particular application.

Access to these ontologies and reasoning over them is provided by the Acuity Controller, which is implemented in Java and built on top of the Jena Semantic Web Framework. Jena provides the backend repository of both concepts (tbox) and instance data (abox). Leveraging Jena capability, model storage can be file-based or use a relational database. ACUITy uses a combination of the Jena in-memory transitive reasoner and special-purpose reasoning embedded in the Acuity Controller. The latter includes default values, learning of defaults from historical instance data, use of scripting to provide behavior, creation of appropriate instances of information objects as required by ontological constraints, and additional behaviors implied by models of the problem/vantage/frame paradigm of work-centered support.

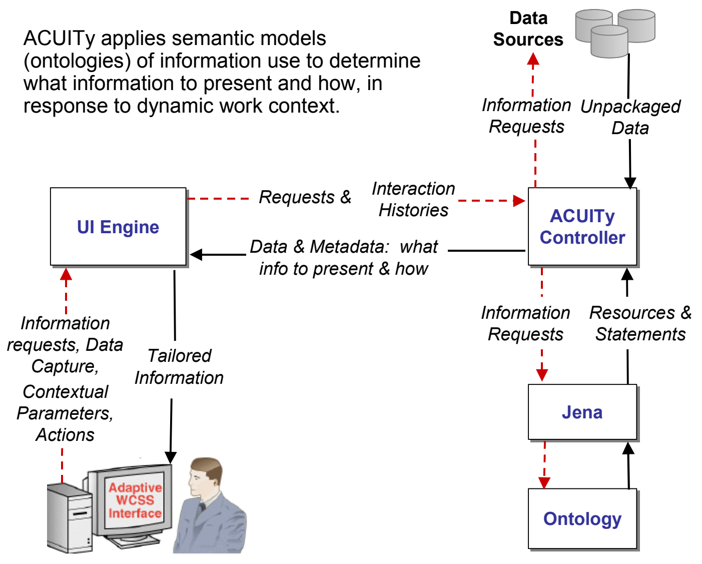

ACUITy applies semantic models (ontologies) of information use to determine what information to present and how, in response to dynamic work context.
Data Sources
Information Requests
Unpackaged Data
UI Engine
Requests & Interaction Histories
ACUITy Controller
Data & Metadata: what info to present & how
Information requests, Data Capture, Contextual Parameters, Actions
Tailored Information
Information Requests
Resources & Statements
Jena
Ontology
Adaptive WCSS Interface

Figure 1. The ACUITy Architecture

Figure 1 shows the ACUITy architecture, consisting of ontologies, the Acuity Controller, the User Interface (UI) Engine, and the adaptive WCSS interface (client).

3.1 The ACUITy Upper-Level and Problem-Vantage-Frame Ontologies

The ACUITy upper-level ontology (AULO) captures very general concepts and is primarily inspired by Sowa's top-level categories (Sowa, 2000). Important upper-level concepts include *abstract thing*[1], with subclasses such as *information object* (from which the user-interface objects described below derive) and *script*. AULO imports the time-entry ontology (Hobbs and Pan, 2004). We expect that more ontologies from Semantic Web Services will be used in the future. Note that we have attempted to follow Rector's (2003) approach to ontology development, which differentiates between "*refining*"entities – entities such as value types and partitioning concepts that are fully dependent on other entities – and "self standing" entities.

User interface design is always based on models of users and the work domain of interest. These can be implicitly understood by the designer or made explicit through a user-centered design process. Explicit implementation models become part of the system itself, usually by hard-coded rules about what to display when. The ACUITy Problem-Vantage-Frame (APVF) ontology captures, in a declarative way, the WCSS framework and essentially offers a way to develop more complete and work-centric models to support user interface design. It also represents interactions within the ACUITy architecture. By explicitly modeling the Problem-Vantage-Frame (PVF) concepts as a central controlling entity in the ACUITy architecture, we better enable adaptive responses to changes in work context and more easily extend user interface designs into new problem domains. Some of the key PVF concepts and their relationships are summarized in Figure 2.

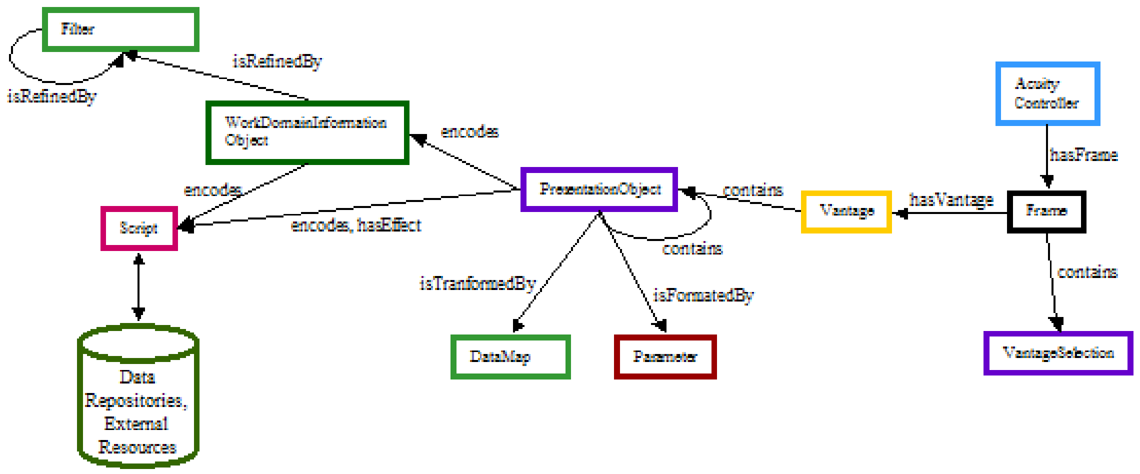


Figure 2: Key Concepts in the Problem-Vantage-Frame Ontology

The APVF ontology defines *presentation object* as an abstract *information object* that describes how to present information to the user. *Presentation objects* have a *presentation nature*, whose range is *display type*, a subclass of *refining*. This range includes 2d graphs, scrolling tables, text (html, plain text, and hyperlinks), display groups, external web applications, and user interaction objects (forms, text entry fields, buttons, and various menu selection mechanisms).

[1] In the remainder of this paper, we will use italics to indicate that a certain word or phrase represents an entity defined in the ACUITy ontology. The word or phrase in italics will not necessarily be exactly the name of the concept in OWL; it will often be a more natural-sounding phrase to the end of making the paper more readable.

A *vantage* 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 the collection of *presentation objects* that are relevant for a particular problem.

Display objects are a subclass of *presentation objects* that *encode* work domain information for presentation to the user. The exact instructions for retrieving information, whether from the ontology or from an external data repository, is captured in a *script*.

Interaction objects, a subclass of *display objects*, allow the user to take action with consequence or side effect, either in the client user-interface or on the server. In the Web interface, client-side effects are implemented as client-side JavaScript. Server-side *scripts* are discussed below. Note that the principles of WCSS encourage a mixed-initiative interface in which the user may take many actions to access information useful to the work at hand. A particular state of the user-interface may have many *interaction objects*.

A property of a *presentation object* that affects its look and feel is referred to as a *presentation parameter*. Unless restricted by the model, *presentation parameters* are accessible to the user at run time for customization

A *frame* represents the various properties of a session and will have one or more *vantages*. While a *frame* may have many *vantages,* only one will typically have the “focus” while the others are accessible in the periphery. This allows the user to work on a problem set that requires different and discretionary insight into the problem domain without losing the overall frame of reference.

The APVF ontology contains these and other concepts and properties which may be used and extended to meet the needs of domain-specific applications. Some of these will be discussed in greater detail in subsequent sections in the context of their primary significance to the overall architecture and functionality.

3.2 Capturing Behavior in Scripts

Borrowing from definition in Sowa’s (2000) upper-level lattice, a script is an abstract form that represents a sequence of instructions. An SQL statement, an RDQL or SPARQL query, the callable statement of a database stored procedure, a client-side Javascript method, and a segment of Java code are examples of different types of scripts. In general, instances of ACUITy scripts are created in the ontological model for the purpose of obtaining information for some purpose, e.g., for encoding in a display object, or for implementing some side effect. Possible side effects include modifying the ontology or an external database and executing some Java code.

Wherever possible, we have used standard script representations (e.g., SQL, SPARQL). With one notable exception, we have found available scripting languages to be adequate to our application needs. Where we have felt an acute unmet need is in the area of ontology update. Just as SQL includes INSERT, UPDATE, and DELETE, we would like to be able to easily specify side effects that act to modify the ontology. To meet this need we have extended RDQL to create eXtended RDQL (XRDQL). Four types of statements have been added with the syntax shown below.

- CREATE ?i WHERE (?i, <rdf:type>, <class>)
- INSERT (<subject>, <predicate>, <object>)
- DELETE (<subject>, <predicate>, <object>)
- UPDATE (<subject>, <predicate>, <object>)

3.3 Anchors

It is often the case that the text of a *script* will need to reference values which are not known at design time. For example, the APVF ontology defines the class *frame*, but a specific Individual of that class is only created when the user instantiates an actual user-interface. RDQL or SPARQL *scripts* can provide a way to reference at run time particular Individuals in the model abox by relation, including type. Reference to an instance of a *script* within another *script* is called an anchor. An anchor is referenced in a *script* by placing the anchor name inside ${…} delimiters. Any named *script* can be used as an anchor. At run time, the anchor will be replaced by the value(s) returned by the referenced *script*. Several "built-in anchors" are known to the Acuity Controller without definition including:

Anchor Name	**Description**
currentAC	The current instance of *AcuityController*
currentUser	The instance of *person* who is identified as the user associated with this *AcuityController* instance
useAnswer	The value or values supplied by the client interface to the effect processing of the corresponding *interaction object*

As an example, one might define the *script* "currentFrame" as instruction to identify the *frame* associated with the current instance of *AcuityController* (the actual association of the *script* text to the Individual *currentFrame* is *hasXRDQLStatement*):

```
currentFrame = SELECT ?f WHERE (${currentAC}, <apvf:hasFrame>, ?f)
```

4 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 and implements certain kinds of specialized reasoning over this knowledge base. In other words, the Acuity Controller queries the ontology to understand where to find data, how to obtain it, how to bundle it, and to perform various computations over the ontology and data. The controller also accepts inputs from the client UI Engine (and possibly other sources) and updates the ontology and/or accomplishes other side effects as modeled in the ontology.

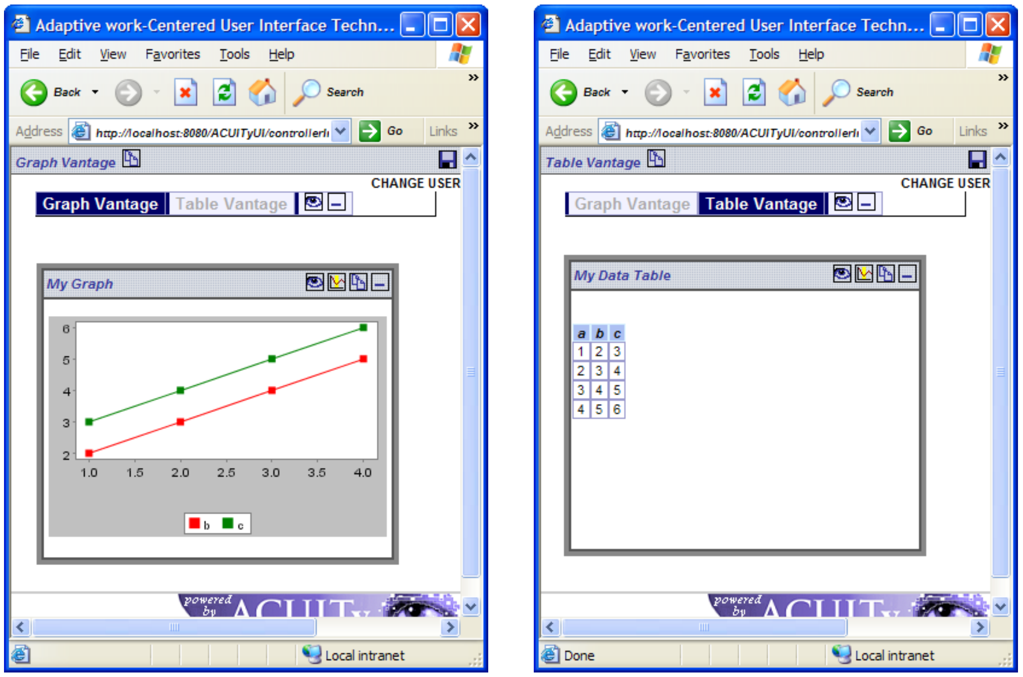
The Acuity Controller, in conjunction with the APVF ontology, provides several implicit behaviors useful to creating flexible, adaptable, extensible work-centered support applications. These include the creation of content according to the restrictions of the domain models, the solicitation of "missing properties," when specified, the monitoring of the parameters of *presentation objects*, and the learning of default values from existing instance data.

4.1 Auto-Generation of Content

A domain ontology will extend the APVF ontology to define domain-specific types of *frames*, *vantages*, and *presentation objects*. These new classes will be defined by various ontological restriction. As a simple example, suppose that a domain application has a single source of data and that we wish to define two types of *vantage*, one containing a tabular display of the data and one showing the data as a graph. We would create the entities shown in Table 1 in our domain ontology.

Table 1: Entities Created to Implement Simple Graph and Table Example

Name	**Type**	**Properties, comments**
MyDataScript	instance of type *script*	query to retrieve domain data, e.g., SQL SELECT statement
MyDataTable	OWL class, subclass of *simple data table*	*encodes* MyDataScript (hasValue restriction), table column headings can be taken from source metadata
MyGraph	OWL class, subclass of *graph*	*contains* MyDataSeries (someValuesFrom restriction)
MyDataSeries	OWL class, subclass of *simple data series*	*encodes* MyDataScript (hasValue restriction), first data column is x-axis (*simple*), each additional data column is a data series. Metadata provides labels
MyTableVantage	OWL class, subclass of *vantage*	*contains* MyDataTable (someValuesFrom restriction)
MyGraphVantage	OWL class, subclass of *vantage*	*contains* MyGraph (someValuesFrom restriction)
MyFrame	OWL class, subclass of *frame*	*has vantages* MyTableVantage and MyGraphVantage (someValuesFrom restrictions); *Acuity Controller* has an added restriction that *has frame* has someValuesFrom from class MyFrame



a	b	c
1	2	3
2	3	4
3	4	5
4	5	6

Figure 3: Client Interface for Simple Graph and Table

Creation of this one Individual and six OntClasses is sufficient to create a working application. A new instance of *AcuityController* will be missing the required property *hasFrame* (see Missing Properties below). When a new instance of MyFrame is created, new instances of MyTableVantage and MyGraphVantage will be created as well because of the someValuesFrom restrictions of MyFrame and because *contains* is a subclass of the *auto instantiate* property in APVF. Likewise, each new *vantage* instance will have its content created as required to meet the model restrictions. When a *vantage* is made the focus vantage, possibly through interaction with the vantage selector, MyDataScript is executed to obtain the data, which is rendered to the display in the end-user client. Figure 3 shows the two *vantages* with a small data set.

We create new classes for the *frame*, *vantages*, and *display objects* because each session created by each user will have its own properties, such as position, size, color, etc., which the user may change. For a user's changes to only affect that session of that user, each session must have its own unique instance data. MyDataScript, by contrast, is the same for all users in all sessions. Therefore we may create it as an Individual.

4.2 Missing Properties

When the necessary conditions of an OWL class indicate that it must have a property but such a property is not found in the ontology for an instance of that class, the property is a "missing property." By making properties subproperties of particular APVF properties, an application developer may cause the AcuityController to take certain actions to resolve missing properties. We already observed in the previous section that *contains* is a subproperty of *auto instantiate*, and therefore if a *presentation object* is missing content the AcuityController will automatically create instances of that content. Another property class in APVF which may be subclassed is the *ask user* property. For example, an instance of *acuity controller* is restricted to have the property *has frame* with a value of type *frame*. Since *has frame* is a subproperty of *ask user*, absence of a *has frame* statement, which will always be the case for a new session, will result in the creation of a temporary *interaction object* offering the user a list of frame instances (previoius sessions) for association with the controller. In fact, since *has frame* is a subproperty of *ask user include create new*, the user's choices will include the option of creating a new *frame*.

4.3 Default Values

The use of OWL as a constraint language for specifying the design of a WCSS application requires the ability to specify some kind of default value. For example, if we are to say that a particular type of *graph series* is a *line* with color *green* but allow the user to change the data series to a *vertical bar* with color *red*, we cannot use ontological restrictions. We need someway to say that the defaults are *line* and *green*.

Since the current specification of OWL does not support default values, we use the annotation property *rdfs:seeAlso* with a value which is an instance of type *default value*. When creating new instances of classes, the Acuity Controller looks for *rdfs:seeAlso* properties with *default values*. If present and if not otherwise restricted, the seeAlso property value is used to provide an editable *presentation parameter* value.

4.4 Learning Defaults and Patterns

The opportunity to learn from accumulated instance data is an implicit benefit of the semantic modeling approach taken by ACUITy. As users adapt the content and visual characteristics of the information that they view in particular problem-solving settings, these changes are stored in the ontology with their context. This past history creates the opportunity for a reasoner to infer what information content (visualizations) is most appropriate based on new information that was unavailable during the initial design of the user interface. The special purpose reasoner then uses this instance data to learn both default content and appearance for new sessions with similar contexts.

Learning can occur at different levels. A single user's preferences can be learned from that user's past behavior. When user models include grouping of users according to role or other shared attributes, learning can occur across peer groups. Sample size and predominance thresholds can be used to control the conditions under which learned values are used. Role-based learned defaults allow new users to benefit from the past behavior of more experienced users.

Learning a default from instance data is the recognition of a simple pattern. While not yet implemented in ACUITy, peer group learning can provide the basis of establishing "best practice" information displays. Recognizing beneficial patterns of usage across groups of users can lead to new classes of *display objects* made explicitly available to developers and users, whereas learned defaults are only implicitly available. This learning of 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. Learning is a powerful tool in making decision support systems adaptable and extensible. It is not inconceivable that abstraction of useful patterns might even extend across application domains.

5 The ACUITy User Interface Engine

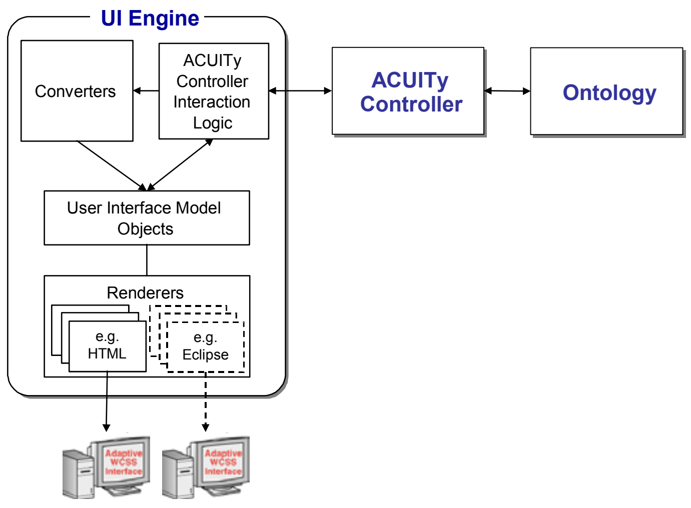


Figure 4. The ACUITy UI Engine

The role of the User Interface (UI) Engine is to accept the ontological information obtained from the Acuity Controller and render the application's user interface. As shown in Figure 4, it sits between the end client and the Acuity Controller and is meant to be the single interface point between the two entities. As such it interacts with the controller to request information from the ontology and instruct the controller to update information in the ontology in response to the user's actions.

It generates user interface display objects to represent the ontological information and data retrieved from the Acuity Controller. The UI Engine itself is made up of a UI controller and UI renderer components. The UI Controller recieves client platform agnostic model objects from the Acuity Controller. The UI renderer then renders the objects to its specific type of client interface. At this point we have implemented a web client renderer to produce a well-formed HTML document from the UI Engine. We expect to build an Eclipse renderer to return Eclipse objects for the interface. This design allows us to add new client interface environments to the overall

ACUITy environment with minimal impact on the applications deployed in the ACUITy environment.

6 ACUITy Benefits

6.1 What ACUITy Means to Users

In an ACUITy application the users themselves finish the design by deciding what information they require to solve a particular problem – defining the *vantage* they need on the problem domain – and changing the characteristics of the information display in order to interact with the data more effectively. This type of direct adaptation is regarded as a critical need in open-ended, information intensive work (Vicente, 2000), but something that is typically difficult to achieve in static information displays. Desktop spreadsheet users are very creative in their adaptations, but distributed spreadsheets have the problem of distributed, inconsistent inputs and distributed results. There is no easy way to aggregate the collective wisdom of user experience. ACUITy captures in a centralized way the experience of users in open-ended problem-solving domains as they gather information from many disjoint sources not precisely identified at design time.

The user can reconfigure the display by adding and removing visualizations defined in the application. The approach can be extended to permit ad-hoc additions of information sources. In particular, Semantic Web Services will allow automatic and/or user-facilitated discovery and inclusion of new sources of relevant information. We would like users to be able to map to these sources of data without architecture or UI re-design. Customization of information display includes, but is not limited to, the hiding, ordering, and sorting of data table columns, the selection of graph series types, e.g., line versus bar, color, and labels, and the type of enumerated selection lists, e.g., dropdown list versus checkbox versus tabs. The user can duplicate and then modify visualizations as desired. Information in disjoint tables and graphs can also be brought into relation by the creation of shared highlight regions, similar to data brushing in statistical graphics.

6.2 What ACUITy Offers to Application Developers

The ACUITy problem-vantage-frame ontology provides developers of new applications a starting point from which they can create information-rich displays by relatively simple model extensions. With respect to the user-interface and information content, the developer is also "finishing the design." For example, a new data table can be added to a display by 1) defining a new subclass of data table, 2) specifying a script, e.g., SQL query, to obtain the table data, and 3) asserting a defining restriction on the new table class that it must "encode" the information returned by the script. All other behavior and attributes necessary for the table to be constructed and displayed, as well as those that allow the user to customize the table display according to their particular preferences, are inherited.

Domain-specific work models will also utilize and extend upper-level ontologies. These ontologies define concepts of time, physical versus abstract, problems, fleets, scripts, processes, and remote data sources. Scripting capability includes support of custom Java code that can implement data access, data transformation, or side effects. This facilitates integration of ACUITy applications with existing information repositories and computational models.

7 Conclusion/Future Directions

Our experience developing several applications is that ACUITy is a powerful and very flexible environment for developing decision support systems. Its ability to allow users to customize their interfaces and to learn their preferences appears to be very promising. Additional user testing and developer feedback is needed to validate this assessment.

ACUITy's flexibility also brings with it the challenge of managing the complexity to which the user is exposed. Ideally, this complexity will be unobtrusive, transparent to the user until additional capability is needed. And, of course, how to access the additional capability should be intuitive. These problems are not unique to ACUITy. As we make the project Open Source, we hope that it will benefit from the innovations of many.

We are starting to create an integrated ACUITy Editor, which will provide considerably more help in application creation and maintenance that does a general-purpose editor such as Protégé-OWL. It will allow application developers and/or users even more power to "finish the design" by creating new classes of presentation objects, ad hoc data queries, etc. In the future we anticipate that new sources of data, e.g., via Semantic Web Services, 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 in real time.

Finally, ACUITy creates data-rich models populated with semantically tagged information that will provide an important resource for further research in user-interface and decision support system design, adaptation, and learning.

8 Acknowledgements

This work was partially funded by the Air Force Research Laboratory, Wright Patterson Airforce Base, under contract F33615-03-2-6300. We would like to acknowledge the contributions of AFRL and Lockheed Martin Simulation and Training Systems, who provided support to and feedback on this research.

References

[1] Eggleston, R., and Whitaker, R. Work-Centered Support Systems Design: Using organizing frames to reduce work complexity, *Human Factors and Ergonomics Society 46th Annual Meeting*, Human Factors and Ergonomics Society, (2002), 265 - 269.

[2] Eggleston, R.G., Young, M.J., and Whitaker, R.D. Work-Centered Support System Technology: A New Interface Client Technology for the Battlespace Infosphere. *National Aerospace and Electronics Conference 2000*, 499-506.

[3] Hobbs, Jerry R. and Feng Pan. 2004. An Ontology of Time for the Semantic Web. *ACM Transactions on Asian Language Processing (TALIP): Special issue on Temporal Information Processing*, Vol. 3, No. 1, March 2004, pp. 66-85. http://www.isi.edu/~pan/time/pub/hobbs-pan-TALIP04.pdf

[4] 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

[5] Sowa, John F. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks/Cole, Pacific Grove, CA (2000).

[6] Vicente, Kim J. HCI in the Global Knowledge-Based Economy: Designing to Support Worker Adaptation, *ACM Transactions on Computer-Human Interaction*, Vol. No. 2 (2000), 263-280.