LA-UR- 98-1065

Approved for public release; distribution is unlimited.

Title: Using multiple perspectives to suppress information and complexity

CONF-980412--

Author(s): Kelsey, Rob-XCM

Webster, Robert-XCM

Hartley, Roger-NM State University

Submitted to: SPIE Conference
Orlando, Florida
March 1998

Los Alamos
NATIONAL LABORATORY

DISCLAIMER

Using multiple perspectives to suppress information and complexity

R. L. Kelsey[a,b] and R. B. Webster[a] and R. T. Hartley[b]

[a]Los Alamos National Laboratory, XCM MS-F645, Los Alamos, NM 87545

[b]New Mexico State University, Computer Science Department, Las Cruces, NM 80003

ABSTRACT

Dissemination of battlespace information involves getting information to particular warfighters that is both useful and in a form that facilitates the tasks of those particular warfighters. There are two issues which motivate this problem of dissemination. The first issue deals with disseminating pertinent information to a particular warfighter. This can be thought of as information suppression. The second issue deals with facilitating the use of the information by tailoring the computer interface to the specific tasks of an individual warfighter. This can be thought of as interface complexity suppression.

This paper presents a framework for suppressing information using an object-based knowledge representation methodology. This methodology has the ability to represent knowledge and information in multiple perspectives. Information can be suppressed by creating a perspective specific to an individual warfighter. In this way, only the information pertinent and useful to a warfighter is made available to that warfighter. Information is not removed, lost, or changed, but spread among multiple perspectives.

Interface complexity is managed in a similar manner. Rather than have one generalized computer interface to access all information, the computer interface can be divided into interface elements. Interface elements can then be selected and arranged into a perspective-specific interface. This is done in a manner to facilitate completion of tasks contained in that perspective. A basic battlespace domain containing ground and air elements and associated warfighters is used to exercise the methodology.

Keywords: information dissemination, information suppression, multiple perspectives, object-based knowledge representation

1. INTRODUCTION

The amount of battlespace data and information generated by reconnaissance, surveillance, and intelligence systems can be overwhelming[1] and continues to increase with the increase of radar sensor systems.[2] The Battlefield Awareness and Data Dissemination (BADD) system is designed to address and solve the problem of too much data and information. Data disseminated in a smart push mode is based on pre-defined profiles. Profiles designate what data and information a particular warfighter needs.[3]

Profiles can be implemented with the notion of perspective. An entity can possess a personalized, limited, and focused view of other entities in a domain. In the case of the battlespace, a particular warfighter possesses a perspective of the other warfighters and warfighting equipment that directly concern that particular warfighter. The perspective is not necessarily static. As conditions and tasks in the battlespace change, so shall the perspective.

Another problem that effects all computer software use is the problem of complex user interfaces. User interfaces are often designed for a wide spectrum of users, from beginner to expert. This can so generalize the interface that it is unfamiliar to many of the users and thus, difficult to use. Like the dissemination of data and information, perspective can be applied to deliver a specialized user interface to a particular user. A perspective generated interface can be personalized and tailored to each user.

(Send correspondence to R.L.K)
R.L.K.: E-mail: rob@lanl.gov
R.B.W.: E-mail: robw@lanl.gov
R.T.H.: E-mail: rth@cs.nmsu.edu

The following sections describe a methodology for object-based knowledge representation which supports the notion of perspective and is implemented in Standard Generalized Markup Language (SGML). Perspective in particular is exploited in creating a framework for representing the entities of the battlespace and representing individual warfighter profiles. An example of specializing and tailoring user interfaces to particular users is also discussed.

2. OBJECT-BASED KNOWLEDGE REPRESENTATION

2.1. Object decomposition

In terms of designing complex software, decomposition is the act of dividing (the whole software) into smaller parts and refining each part independent of the others.[4] Decomposition lessens complexity. Object (based, oriented) decomposition occurs when the parts are objects. Booch[4] refers to objects as "key abstractions in the problem domain." Objects are tangible entities which model something in the real world and embody and exhibit their own behavior.[4]

Creating a representation of a domain of knowledge is not unlike designing complex software. The process of abstraction and decomposition is much the same. The battlespace domain consists of tangible entities. Each of the entities models something from the real world and exhibits unique and individual behavior. For these reasons, object decomposition is chosen to abstract and decompose the example domain for the purpose of creating a representation.

2.2. Representation using objects

Representation using objects means that representations are designed and creating using object constructs. In the methodology, the constructs include the class, domain, and instance.

A class represents a collection of objects. It can be considered a template for creating an instance or object. A class construct has the components attributes and methods. An attribute is a characteristic or property of the objects of the class. A method represents a behavior. It is an operation on an object of the class.

A domain defines a group of instances or objects from the available classes. An instance represents an actual and unique object as defined by a class. An instance is instantiated from a class into the domain and (at instantiation) values are assigned to its attributes.

There are also a number of relationships between these constructs that are supported. The relationships are inheritance, aggregation, perspective, and content-specific. With inheritance, a class can be created to be a child of another (parent) class. Thus, the child class inherits all the parent's attributes and methods. With aggregation, a class can be created that is made of components and each component is a class. With perspective, an object of a class can possess a personalized, limited, and focused view of other objects (of classes) in the domain.

A content-specific relationship occurs between two objects (of classes) when one object operates on another object. That is, one object calls a method of another object. These relationships depend on the nature and content of the method.

2.3. SGML implementation

2.3.1. Introduction and background

The Standard Generalized Markup Language (SGML) is an ISO standard for the description of documents.[5] This standard describes technology for facilitating text interchange in documents,[6] but the technology has many potential uses. SGML is used to describe the structure and organization of a document, or more specifically, the structure and organization of the content of a document. As a result, the format of the content of a document and the content of a document are two separate things. In short, SGML enables the design and implementation of document structure and the associated notation for marking the content of a document.

Although SGML was originally designed for the publishing of both paper and electronic documents, it has been exploited for use in other areas including enterprise management, engineering, product life cycle, and multimedia. Any application that must process and manage information may benefit from SGML. SGML can help enable applications that use indexing and retrieval of information, cross reference of information, modification and revision of information, and multiple display formats.

There are three components or parts to an SGML document.[7] The first component is the SGML declaration which determines the formal syntax and any optional features to be used within documents. These are the rules

for designing a document type definition (DTD) and for validating the conformance of a DTD and an associated document.

The second component is the document type definition (DTD) which is a rule set for a class or group of documents. The rules in a DTD describe the notation for marking the content pieces of a document and how those pieces relate to one another. Some example classes of documents are project plans, memos, and technical reports. Each of these classes of documents has its own style and format.

The third component is an instance of a document which is an actual document containing content and markup. Implementing the methodology in SGML involves creating a DTD that maps the methodology constructs and their parts into SGML constructs.[8] An actual representation of knowledge is what would be a document instance.

2.3.2. Reasons for using SGML

There are a number of reasons for using SGML to implement a knowledge representation scheme. SGML provides benefits that are not normally available in other representation schemes. One benefit is that a DTD is a formal definition of the representation constructs and how those constructs relate to one another. This means that a representational structure is provided. Thus, the DTD as a grammar and with an SGML parser can enforce the representational structure.

Another benefit is that SGML is a markup language and there is growing familiarity with markup languages and their use. Finally, SGML is portable. A document instance can be ported and translated to other systems and other formats by creating post-parser applications.

2.4. Constructs implemented in SGML

The class construct is shown in Figure 1. This is the template and markup used to define a class. Within a class are any number of attributes and methods. Shown within the class construct are the templates and markup for attributes and methods. Name identifies the class, attribute, or method, respectively. The content tags within a method contain the actual content of an operation. This could be a rule, pseudocode, or executable code.

```
<class name="  ">
  <attribute name="  ">
  </attribute>
  <method name="  ">
    <content>
    </content>
  </method>
</class>
```

Figure 1. Template and markup for a class construct.

The domain construct is shown in Figure 2. This is the template and markup used to define a domain. Within a domain are instances of the previously defined classes. Shown within the domain construct is the template and markup for an instance. Name identifies the domain or instance, respectively. Classname identifies the class from which the instance is instantiated.

```
<domain name="  ">
  <instance classname="  "   name="  ">
  </instance>
</domain>
```

Figure 2. Template and markup for a domain construct.

A perspective is usually defined within a class construct. Figure 3 shows the template and markup used to define a perspective from within a class construct. Name (for perspective) identifies the perspective. Within the def tags is the definition of the perspective which can suggest context. An iclass is a class that is included within the perspective. That is, objects of this class are seen within the perspective. Name (for iclass) identifies the included class. Within

```
<perspective name="  ">
  <def></def>
  <iclass name="  ">
    <atts></atts>
    <mets></mets>
  </iclass>
</perspective>
```

Figure 3. Template and markup for a perspective construct defined within a class.

the atts tags is a list of included attributes of the included class. Within the mets tags is a list of included methods of the included class. There can be any number of included classes.

A perspective that is defined or further defined within an instance construct has additional notation. Figure 4 shows the additional template and markup for definition of a perspective within an instance construct. Name (for focus-perspect) identifies the newly defined or further defined perspective. Name (for perval) identifies an included class or included attribute (if a nested perval) in the perspective. Methods (for perval) is a list of included methods of the included class. Within the perval tags can be a name of an instance of an included class or the value of an included attribute.

```
<focus-perspect name="  ">
  <perval name="  "   methods="  "></perval>
</focus-perspect>
```

Figure 4. Template and markup for perspective defined within an instance construct.

3. BATTLESPACE DOMAIN

3.1. Introduction

The battlespace domain consists of knowledge about the entities in a battlespace and how those entities interact with one another to model a battlespace. The entities that participate in a battle include people and equipment and are described in the following section. A larger and more complex version of this domain is being used for the Defense Advanced Research Projects Agency (DARPA) High Performance Knowledge Bases (HPKB) project.[9]

3.2. Battlespace entities

The flying observer flies around (out of harm's range) over the land portion of the battlespace observing activity on land. The flying observer attempts to identify targets (both moving and stationary), track the targets (if moving), and classify the targets as friend or enemy and what they are (person, equipment, et cetera). The flying observer will pass this information along to those who can make use of the information (on the same side). A sighting is recorded information of a potential target of interest sighted by the flying observer. Sightings are updated as necessary and passed to interested parties.

An unmanned aerial vehicle is a flying drone that is piloted remotely by a human controller or pilot. The unmanned aerial vehicle is used to make close proximity observations of sightings made by the flying observer. It is unmanned because of the high potential for harm when making its observations. A missile launch site is a ground site and equipment capable of launching missiles at aerial targets. There are two types of missile launch sites: fixed and mobile. A fixed site cannot be moved. A mobile site can be moved, but can only launch missiles when stationary and in a setup position.

A battalion is a large armed force which can consist of soldiers and vehicles as elements. A battalion moves about the land portion of the battlespace. A vehicle is any of a number types used in battle, such as trucks, tanks, and troop transports. A truck can carry equipment and supplies. A tank is armored and armed. A troop transport transports soldiers. Vehicles move about the land portion of the battlespace. A civilian is a person who is a non-combatant and unarmed. Civilians may be moving about the land portion of the battlespace.

3.3. Battlespace domain SGML representation

Figure 5 shows the definition for the class flying-observer. The class flying-observer has the attributes bearing, position, velocity, and field-of-view and the methods update-position, change-bearing, change velocity, and issue-sighting. The contents of the methods are not shown. Bearing is a compass direction in degrees. Position is a three-dimensional coordinate. Velocity is miles per hour. Field-of-view is a radial distance in miles.

```
<class name="flying-observer">
  <attribute name="bearing"></attribute>
  <attribute name="position"></attribute>
  <attribute name="velocity"></attribute>
  <attribute name="field-of-view"></attribute>
  <method name="update-position"></method>
  <method name="change-bearing"></method>
  <method name="change-velocity"></method>
  <method name="issue-sighting"></method>
</class>
```

```
<class name="sighting">
  <attribute name="id"></attribute>
  <attribute name="timestamp"></attribute>
  <attribute name="prob-position"></attribute>
  <attribute name="prob-classification"></attribute>
  <method name="update"></method>
</class>
```

Figure 5. The definitions for the class flying-observer and the class sighting.

Also shown in Figure 5 is the class sighting. The class sighting has the attributes identification (id), timestamp, probable position (prob-position), probable classification (prob-classification) and the method update. The contents of the method are not shown. Id is a unique identifying symbol or number of a particular sighting. Timestamp is a date and time reference. Prob-position is the best estimate for position of the sighting. Prob-classification is the best estimate of classifying the sighting.

Figure 6 shows the classes unmanned aerial vehicle (u-aerial-vehicle) drone, and ground-pilot. The class u-aerial-vehicle is an aggregation of the classes drone and ground-pilot. This is the reason why the class u-aerial-vehicle has attributes drone and ground-pilot and the attributes contain instances of the respective classes. The class drone has attributes bearing, position, velocity, and field-of-vision and methods update-position, change-bearing, and change-velocity. The class ground-pilot has attribute position. The type of values for these attributes is the same as for the class flying-observer.

```
<class name="u-aerial-vehicle">
  <attribute name="ground-pilot">
    <instance classname="ground-pilot">
  </attribute>
  <attribute name="drone">
    <instance classname="drone">
  </attribute>
</class>

<class name="drone">
  <attribute name="bearing"></attribute>
  <attribute name="position"></attribute>
  <attribute name="velocity"></attribute>
  <attribute name="field-of-view"></attribute>
  <method name="update-position"></method>
  <method name="change-bearing"></method>
  <method name="change-velocity"></method>
</class>

<class name="ground-pilot">
  <attribute name="position"></attribute>
</class>
```

```
<class name="missile-launch-site">
  <attribute name="position"></attribute>
  <method name="launch-missile"></method>
</class>

<class name="stationary-launcher" inherit="missile-launch-site">
</class>

<class name="mobile-launcher" inherit="missile-launch-site">
  <attribute name="configuration"></attribute>
  <method name="change-config"></method>
  <method name="move-launcher"></method>
</class>
```

Figure 6. The definition for class u-aerial-vehicle which is an aggregation of the classes drone and ground-pilot (also shown) and the definitions for classes missile-launch-site, stationary-launcher, and mobile-launcher. The latter two classes inherit from the class missile-launch-site.

Also shown in Figure 6 are the definitions for the classes missile-launch-site, stationary-launcher, and mobile-launcher. The classes stationary-launcher and mobile-launcher inherit from the class missile-launch-site (indicated by `inherit="missile-launch-site"`). The class missile-launch-site has the attribute position and the method launch-missile. This attribute and method are inherited by the classes stationary-launcher and mobile-launcher. The class mobile-launcher has the additional attribute configuration and methods change-config and move-launcher. Position is the same as for the class flying-observer. Configuration refers to whether the launcher is setup for movement (mobile) or for launching missiles (launch).

In Figure 7 is shown the definition for the class battalion. As shown, the class battalion has the attribute position. Normally, there would be other attributes depending on the aggregation of other classes including vehicle and soldier. Also shown in Figure 7 are the definitions for the classes human, soldier, and civilian. The classes soldier and civilian inherit from the class human. The class human has the attributes name and position and the method change-position. These attributes and method are inherited by the classes soldier and civilian. The class soldier has the additional attribute rank. Name is a unique name identifying a person. Position is the same as for the class flying-observer. Rank is military rank.

```
<class name="battalion">
  <attribute name="position"></attribute>
</class>


<class name="human">
  <attribute name="name"></attribute>
  <attribute name="position"></attribute>
  <method name="change-position"></method>
</class>

<class name="soldier" inherit="human">
  <attribute name="rank"></attribute>
</class>

<class name="civilian" inherit="human">
</class>
```

Figure 7. The definitions for classes battalion, human, soldier, and civilian. The classes soldier and civilian inherit from the class human.

Figure 8 shows the definition for the class vehicle. The class vehicle has the attributes type, size, bearing, position, and velocity and the methods update-position, change-bearing, and change-velocity. Type refers to the kind of vehicle. Size refers to the physical size. Bearing, position, and velocity are the same as the class flying-observer.

```
<class name="vehicle">
  <attribute name="type"></attribute>
  <attribute name="size"></attribute>
  <attribute name="bearing"></attribute>
  <attribute name="position"></attribute>
  <attribute name="velocity"></attribute>
  <method name="update-position"></method>
  <method name="change-bearing"></method>
  <method name="change-velocity"></method>
</class>
```

Figure 8. The definition for the class vehicle.

3.4. Potential perspectives in the battlespace

The main purpose of identifying and creating multiple and varying perspectives of the battlespace is to break up the large amount of available information into chunks of a more usable size. Within a perspective, not only is the

amount of information smaller, but it should also be of more interest to the possessor of the perspective. Potential perspectives can be based on individual tasks, proximity of position of forces, and posture of forces.

A perspective based on a task consolidates the entities that contribute to or are needed for completing the task. For example, a missile launcher is interested in aerial targets. Therefore, classes of aerial vehicles (such as flying-observer and drone) are included in a missile launcher's perspective. Another example is the task of a field commander to search out and destroy missile launch sites. The class of missile launch sites is included in the commander's perspective.

A perspective can be further focused by specifics to a task. Consider a flying-observer whose job it is to observe activity on land and identify and track potential targets. This suggests a perspective that includes all classes of the land portion of the battlespace, but in reality, a flying-observer can only detect targets within a specified range (field-of-view). Thus, the perspective must include the position attribute of all included classes and constrain the value of position to within the value of field-of-view.

The class drone is another class with a task and position based perspective. A drone makes close proximity inspections of flying-observer sightings. Within the drone's perspective is the ground-pilot (who controls the drone) and anything on land within the drone's field-of-view. The ground-pilot has a perspective which includes the drone and the flying-observer (who dispatches the unmanned-aerial-vehicle).

A perspective can also be further focused by the posture of the forces. A field commander in charge of defending an area may only be interested in an opposing force if it is attacking. Posture may be inferred from the position and bearing of an opposing force.

3.5. An example scenario for perspective

Consider a command hierarchy to illustrate the notion of perspective. A commander of several tank battalions views these elements as battalions and not as individual tanks. As a result, the position, posture, and status of each battalion takes precedence over the same characteristics of each individual tank. The commander possesses a coarser-grained view of the elements. Alternatively, an operator of a tank (within one of the battalions) views the other tanks. These two different grained views of the battlespace can be represented using perspectives.

Whereas the commander is concerned with the logistics of engagement on a larger scale (location and movement of entire battalions), the tank operator is concerned with things like location of companion tanks and identifying and locating enemy tanks and targets. These different perspectives can be supported within the representation of the battlespace. Figure 9 shows the definition of a class commander which inherits from the class soldier and contains the perspective tank-command. The perspective tank-command includes the class battalion (indicated within `iclass` tags). Also shown is the definition of a class tank-operator which inherits from the class soldier and contains the perspective tank-operator. The perspective tank-operator includes the class tank (not previously defined). These perspectives are each defined in the class which will have objects (of the class) possessing the perspective.

```
<class name="commander" inherit="soldier">
  <perspective name="tank-command">
    <def>command level view of tank battalions</def>
    <iclass name="battalion"></iclass>
  </perspective>
</class>

<class name="tank-operator" inherit="soldier">
  <perspective name="tank-operator">
    <def>operator view of tank</def>
    <iclass name="tank"></iclass>
  </perspective>
</class>
```

Figure 9. The definitions for class commander and class tank-operator and their perspectives.

A perspective can be further focused by identifying instances (as opposed to classes) that are included in the view. For example, suppose the tank commander was only interested in Battalion A and Battalion B. These specific instances of the class battalion can be included in the commander's perspective, but not in a class definition

(as in Figure 9). Class definitions have no information regarding their instantiations. This means that the focus of the perspective must be defined in the instance construct of the tank commander. Figure 10 shows the declaration of the instance named tank-commander (indicated by name="tank-commander") of the class commander (indicated by classname="commander"). The tank-commander views all other objects (in a domain) through the perspective tank-command (indicated by perspect="tank-command"). The perspective tank-command is further focused with the <focus-perspect name="tank-command">...</focus-perspect> construct. The perval tag lists specific instances (by name) and specific attributes (by value) to further constrain what is viewed within the perspective. The perval tag shown lists the instances named Battalion-A and Battalion-B of class battalion (indicated by name="battalion") that constrain the perspective tank-command. This means that only Battalion-A and Battalion-B (of the class battalion) are viewed within the perspective tank-command.

```
<instance classname="commander" name="tank-commander" perspect="tank-command">
  <focus-perspect name="tank-command">
    <perval name="battalion">
      Battalion-A
      Battalion-B
    </perval>
  </focus-perspect>
</instance>
```

Figure 10. The declaration of instance tank-command of class commander.

3.6. Maintaining and changing perspectives

A perspective can serve as an information profile for the possessor (owner) of the perspective. With such a profile, it can be determined what information should be disseminated to the owner of the profile. Irrelevant information will be suppressed and the owner will not be overwhelmed. Also, the owner will not have to sort through information looking for the pertinent pieces.

As a task changes, so will a perspective. Included classes and included instances of those classes can be added and deleted from a perspective as necessary. This will keep a profile current and more importantly, the correct dissemination of information current. Information can be consolidated, focused, and kept pertinent to a specific task.

4. USER-INTERFACE COMPLEXITY

4.1. Introduction

A complex user interface can be difficult to use and difficult to learn. Often times, an interface is complex because it is designed for a large cross section of users and tasks. This means the interface and tasks have been generalized to accommodate an entire spectrum of users from beginner to expert.

What is needed is a more dynamic user interface. The interface should accommodate different levels of users and change as a particular user learns or the user's tasks change. The interface should be tailored and personalized to each user and the user's tasks.

If a user interface can be decomposed into separate interface entities or components, then it can be represented with the SGML constructs previously discussed. Additionally, perspectives can be created that aggregate interface components into an individual, personal interface.

4.2. Example problem

An example using a portion of a Web-based interface shows how the notion of perspective can contribute to creating an interface tailored to a particular user. Although this example is not specific to the battlespace domain it does illustrate the same techniques and framework that would be used in creating an interface for any application. A typical task in database and Web applications is retrieval of information or knowledge. In this example a collection of documents accessible through a Web server exists. A user accesses the Web server through the use of a Web browser such as NCSA Mosaic[10] or Netscape Navigator.[11] Upon access of the Web server, a hyper-text markup

language (HTML) form is provided for querying the document collection. It is this HTML form that is the portion of the Web-based interface that may be tailored to an individual user and the user's tasks.

In general, the task of retrieval can be characterized in two ways and these characterizations help determine interface entities for the HTML form. In one type of retrieval a user knows exactly what they want and what query to make. In the other type of retrieval the user does not know what they want or may be unsure of what they want. This user does not know what query to make, but may be able to find what they want by browsing and subsequently making a selection. For the purpose of this research the former type of retrieval will be known as *knower* (user knows what they want) and the latter type will be known as *browser* (user browses for what they want).

In the document collection example a user of type *knower* enters a query consisting of the title or identification number of a document. A user of type *browser* selects from a list of document titles. These two types can be thought of as two different perspectives of retrieval. A user possessing the *knower* perspective sees the Web-based interface differently from a user possessing the *browser* perspective. This is because the interface from one perspective to another is indeed different. It is different because it is personalized and tailored to the user based on their perspective. The interface of the *knower* perspective consists of different HTML elements than the interface of the *browser* perspective.

4.2.1. Class design

Four classes are necessary, three of them represent HTML elements and the fourth represents a user of the interface. The first HTML element to represent is for input of text. This is a single text input box which a user with the *knower* perspective can use to request a document. In the representation this HTML element will be the class text-box. The class text-box has the attributes name, heading, and size. The attribute name contains a value of how to refer to the instance of class text-box or what the instance is called (or named). The attribute heading contains a value for a heading or title for an instance of the class text-box. This value could be an instruction for use of the input text box and appears before the instance of the class text-box in an interface. The attribute size contains the value for the size of an instance of class text-box. Size is measured in number of characters and determines the length of an instance of class text-box in an interface. Figure 11 shows the definition of class text-box.

```
<class name="text-box">
  <attribute name="heading"></attribute>
  <attribute name="name"></attribute>
  <attribute name="size"></attribute>
  <method name="initiator">
    <content>
    print "<h2>";
    print "$attrib_vals{'heading'}</h2><p>\n";
    print "<input name=\"$attrib_vals{'name'}\"";
    print "<size=\"$attrib_vals{'size'}\">\n";
    </content>
  </method>
</class>
```

Figure 11. The definition of class text-box.

In the class text-box there is the method initiator which has method content written in Perl. This method initiates the display of a text box writing the appropriate HTML to standard out. Notice the opening <h2> tag which signifies a level-2 heading and the code that follows obtaining the value of the attribute heading. A post-parser routine makes the parsed knowledge and information available to a method and then evaluates the method. This illustrates a close relationship between design of knowledge (in the representation) and use of knowledge (in the post-parser routine). The object-based methodology and its SGML implementation support this relationship.

The second HTML element to represent is an input selection box. This is a box containing a list of items and (depending on the selection type) one or more items may be highlighted for selection. A user with the *browser* perspective can browse the items in the list and make a selection. In the representation, this HTML element will be the select-box.

The class select-box has the attributes name, heading, size, type, and options. The attribute name contains a value of how to refer to an instance of the class select-box or what the instance is called (or named). The attribute

heading contains a value for a heading or title for an instance of the class select-box. Like the attribute heading for class text-box, this value could be an instruction for use of the selection box and appears before the instance of the class select-box in an interface. The attribute size contains the value for the number of list items visible before scrolling must occur. The attribute type contains the value multiple for multiple item selection or no value for single item selection. The attribute options contains values for the list items that appear in the selection box. Figure 12 shows the definition of class select-box. Like the class text-box, the class select-box has a method named initiator that initiates the display of a selection box by writing the appropriate HTML to standard out.

```
<class name="select-box">
  <attribute name="heading"></attribute>
  <attribute name="name"></attribute>
  <attribute name="size"></attribute>
  <attribute name="type"></attribute>
  <attribute name="options"></attribute>
  <method name="initiator">
    <content>
    print "<h2>";
    print "$attrib_vals{'heading'}</h2><p>\n";
    print "<input name=\"$attrib_vals{'name'}\"";
    print "<size=\"$attrib_vals{'size'}\">\n";
    ...
    print "</select>\n";
    </content>
  </method>
</class>
```

Figure 12. The definition of class select-box.

The third HTML element is the form or fill-out form element. This HTML element actually encapsulates elements like the input text box and input selection box. The text and selection boxes do not work without appearing within the form element. The form element designates what is done with the request from text and selection boxes. In the representation, the form element will be the class html-form.

The class html-form has the attributes action, text-input, and select-input. The attribute action contains the value of where the contents of the form are to be submitted. The attributes text-input and select-input are instances of the classes text-box and select-box, respectively. These attributes are aggregate parts. Figure 13 shows the definition of class html-form. The class html-form has a method named initiator which initiates the display of an HTML form and its associated content (text box or selection box depending on the perspective) by writing the appropriate HTML to standard out.

```
<class name="html-form">
  <attribute name="action"></attribute>
  <attribute name="text-box">
    <instance classname="text-box">
  </attribute>
  <attribute name="select-box">
    <instance classname="select-box">
  </attribute>
  <method name="initiator">
    <content>
    print "<form action=\"$attrib_vals{'action'}\">";
    eval ("@cont");
    print "</form>\n";
    </content>
  </method>
</class>
```

Figure 13. The definition of class html-form.

The last class to define represents a user of the interface and is named generic-user. The class generic-user is an agent class because instances of the class will possess one of the two perspectives *knower* or *browser*. Figure 14

shows the definition for the class generic-user. The two different perspectives are defined in the class generic-user and are indicated by the <perspective ...> tag. The perspective *knower* includes the classes html-form and text-box (indicated by iclass tags) and has the definition "a user with the knower perspective knows exactly what it is they are looking for." The perspective *browser* includes the classes html-form and select-box (indicated by iclass tags) and has the definition "a user with the browser perspective browses for what they are looking for." This means, for example, a particular user who possesses the perspective *knower* will see instances of the classes html-form and text-box. Additionally, the user will see those instances by the attributes that have been designated to be included in the perspective *knower* (indicated by list of attributes within atts tags). Methods are designated to be included in the perspective in the same manner (except indicated by mets tags).

```
<class name="generic-user">
  <attribute name="login"></attribute>
  <perspective name="knower">
    <def>A user with the knower perspective...</def>
    <iclass name="html-form">
      <atts>action
            <iclass name="text-box">
              <atts>heading name size</atts>
              <mets>initiator</mets>
            </iclass>
      </atts>
      <mets>initiator</mets>
    </iclass>
  </perspective>
  <perspective name="browser">
    <def>A user with the browser perspective...</def>
    <iclass name="html-form">
      <atts>action
            <iclass name="select-box">
              <atts>heading name size type options</atts>
              <mets>initiator</mets>
            </iclass>
      </atts>
      <mets>initiator</mets>
    </iclass>
  </perspective>
</class>
```

Figure 14. The definition of class generic-user.

4.2.2. Domain definition

In the example problem a domain is created for use from the defined classes generic-user, html-form, text-box, and select-box. The domain is named webinterface and contains two instances of the class generic-user and one instance each of the classes html-form, text-box, and select-box. Attributes in each of the classes are assigned values at the time of instantiation. Instances of class generic-user (an agent class) are also assigned the perspective in which they view the rest of the domain at instantiation. Figure 15 shows the domain webinterface.

In the domain webinterface one instance of class generic-user is named Smith and the other instance is named Jones. The instance Smith possesses the perspective *knower* (indicated by perspect="knower") and the instance Jones possesses the perspective *browser* (indicated by perspect="browser"). The instance of the class html-form is named interface-form. Each attribute in the instance has an assigned valued as indicated by what appears between the begin and end attval tags. The attribute action has the value http://www.test.gov/cgi-bin/parse-query. The attribute text-input is an instance of the class text-box and thus, values for attributes of class text-box are assigned. This accounts for the use of nested attval tags. The attribute text-input heading has the value Enter Title. The values for the attributes text-input name and text-input size are doctitle and 50, respectively.

The next attribute shown (in Figure 15) is select-input which is an instance of select-box and it too has nested attval tags. The values for the attribute select-input heading, select-input name, select-input size, and select-input

```
<domain name="webinterface">
  <instance classname="generic-user" name="Smith" perspect="knower">
    <attval name="login">smith</attval>
  </instance>
  <instance classname="generic-user" name="Jones" perspect="browser">
    <attval name="login">jones</attval>
  </instance>
  <instance classname="html-form" name="interface-form">
    <attval name="action">http://www.test.gov/cgi-bin/parse-query</attval>
    <attval name="text-input">
      <attval name="heading">Enter Title</attval>
      <attval name="name">doctitle</attval>
      <attval name="size">50</attval>
    </attval>
    <attval name="select-input">
      <attval name="heading">Select A Single Title</attval>
      <attval name="name">doctitle</attval>
      <attval name="size">5</attval>
      <attval name="type"></attval>
      <attval name="options">
        Semantic Networks
        Database Schema
        Frames and Objects
        Logical Reasoning
      </attval>
    </attval>
  </instance>
</domain>
```

Figure 15. The knowledge domain interface containing instances of the classes.

type are `Select A Single Title`, `doctitle`, `5`, and null (no value for type), respectively. The attribute select-input options has for its value the listed titles Semantic Networks, Database Schema, Frames and Objects, Logical Reasoning, et cetera.

4.2.3. Scenario of use

A typical scenario of use of the domain of knowledge is driven by a post-parser routine. In the example problem, a user accesses the document archive with a login name. The post-parser routine checks the domain for an instance of class generic-user with the login name. The perspective possessed by the generic-user instance is determined by the post-parser routine so that the personalized interface can be served. This means the post-parser routine gathers the instances included in the particular perspective and displays them according to the included attributes and methods.

For example, user Smith logs into the document archive using his favorite Web browser. User Smith being a regular and accomplished user of the document archive knows the title of documents he wishes to obtain and this knowledge (Smith has the perspective *knower*) is represented in the domain. As a result, after login, the next HTML page Smith sees includes an HTML input text box. Figure 16 shows a fragment of the HTML page as determined by the perspective (*knower*) Smith possesses.

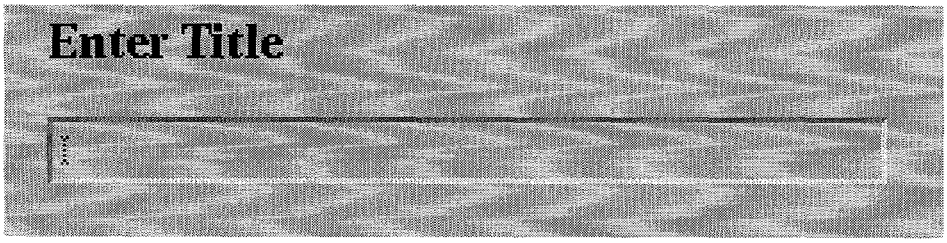

Figure 16. A fragment of the interface within the perspective *knower*.

The sequence of actions for user Jones is the same as user Smith. The difference is that user Jones is less familiar with the contents of the document archive and prefers to select a document from a list of titles. This preference is

reflected in the perspective (*browser*) user Jones possesses. As a result, the HTML page Jones sees includes an HTML input selection box. Figure 17 shows a fragment of the HTML page as determined by the perspective (*browser*) Jones possesses.

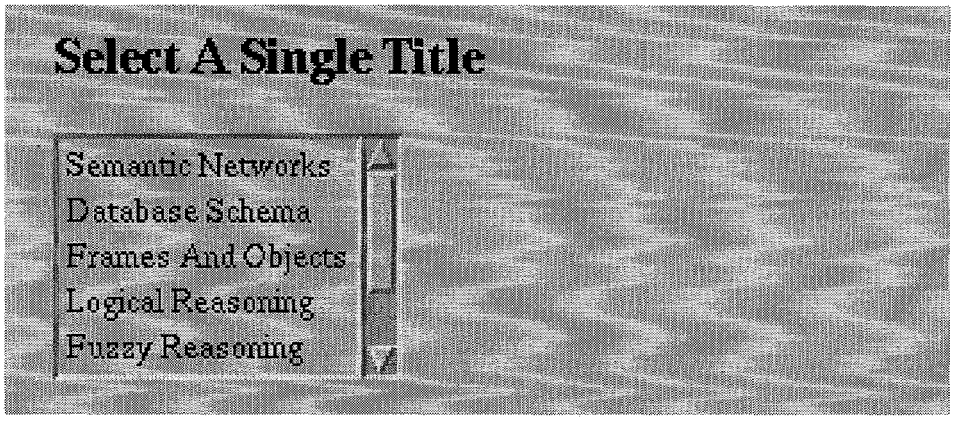

Figure 17. A fragment of the interface within the perspective *browser*.

4.2.4. Changing a perspective

As a user becomes more familiar with a user interface it is often the case that the user becomes more adept and faster at using the interface. This can sometimes lead to the user wanting interface shortcuts, changes in the interface, or an altogether different interface. This is especially true when the user has become so adept to surpass the ease of use of the interface and now finds its use slow and frustrating. The reverse can also be true when an expert user has not used the interface in a long time and is no longer familiar with its use. There can also be changes in a user's task or how that user accomplishes that task that will contribute to potential changes in an interface.

Consider the user Jones who prefers to browse document titles and select from a list. Suppose Jones has become more familiar with the user interface. Further suppose that Jones has also become more familiar with the document archive and now knows the titles of many of the documents of interest. Sometimes Jones finds that scrolling through the long list of documents can be cumbersome. Other times, Jones is glad this list is there when a particular document title cannot quite be remembered. What Jones would really like is to have both an input text box and an input selection box on his interface and the choice of using one or the other. What Jones needs is a change in perspective.

Given the current knowledge domain (shown in Figure 15) and the currently defined perspectives, Jones can easily be accommodated. If the instance Jones of class generic-user possesses both the perspectives *knower* and *browser*, then user Jones will see an interface that has an input text box and an input selection box. The perspective for instance Jones in the domain must be changed or rather re-assigned. The `perspect="browser"` portion of the instance tag (for Jones) is re-assigned to `perspect="knower browser"` to include both perspectives.

Re-assignment of currently defined perspectives is trivial, however, knowing when re-assignment should occur is less trivial. A change in perspective may occur at the request of a user or at the suggestion of an observer agent. Since the user knows nothing of the knowledge representation or terminology, a requested change in perspective is accomplished by questioning the user with basic terminology. For example, the user may be asked whether he/she prefers entering a title or selecting a title from a list. A user may also be shown interface elements and asked which one they prefer. A user may also say "give me an interface like Sam's," given that Sam is an instance of class generic-user in the domain. These are the same types of questions that may be asked when initially configuring a new user and adding them as an instance to the knowledge domain.

A perspective may also be changed at the suggestion of an observer agent and the subsequent agreement of the user. The observer agent watches the frequency, the requests, and ways a user makes requests of use of the document archive. The observer agent attempts to analyze this use to determine if a user would be better suited doing what they are doing with a different interface. A change in perspective is never made without the agreement of the user.

5. CONCLUSIONS

A methodology for object-based knowledge representation and its application towards a framework for the dissemination of battlespace data and information has been discussed. The notion of perspective is supported within the methodology and is particularly well suited for representing individual warfighter profiles. Data and information can be consolidated, suppressed, and focused for an individual's needs. Additionally, a perspective is dynamic and can be changed to fit the changing circumstances and tasks of an individual.

The design and creation of personalized computer user interfaces demonstrates another useful application of perspective. Interface elements can be consolidated into a user interface personalized and tailored to the task-specific needs and expertise level of an individual user. As the tasks change or the expertise level changes, the perspective can change to create another interface.

The implementation of the framework in SGML allows for enforcement of representation structures, familiarity and use of markup languages, and portability. The advance and popularity of the World Wide Web has brought about many applications of SGML including Web-based user interfaces for information dissemination and retrieval. The representation of knowledge, information, and data is a natural extension from SGML's original purpose of document creation and management.

REFERENCES

1. R. J. Douglass, J. Mork, and B. R. Suresh, "Battlefield awareness and data dissemination (badd) for the warfighter," in *Digitization of the Battlefield II*, R. Suresh, ed., vol. 3080, pp. 18–24, SPIE, SPIE, 1997.
2. J. F. Gilmore and R. O. Johnson, "Battlespace exploitation vision," in *Digitization of the Battlefield II*, R. Suresh, ed., vol. 3080, pp. 43–48, SPIE, SPIE, 1997.
3. T. Stephenson, B. DeCleene, G. Speckert, and H. Voorhees, "Badd phase ii dds information management architecture," in *Digitization of the Battlefield II*, R. Suresh, ed., vol. 3080, pp. 49–58, SPIE, SPIE, 1997.
4. G. Booch, *Object-Oriented Analysis and Design with Applications*, Addison-Wesley Publishing Company, Menlo Park, CA, second ed., 1994.
5. International Organization for Standardization, Geneva, *ISO 8879:1986 Information processing - Text and office systems - Standard Generalized Markup Language (SGML)*, October 1986.
6. E. van Herwijnen, *Practical SGML*, Kluwer Academic Publishers, Boston, MA, second ed., 1994.
7. L. Alschuler, *ABCD...SGML A User's Guide To Structured Information*, International Thomson Computer Press, Boston, MA, 1995.
8. R. L. Kelsey, R. T. Hartley, and R. B. Webster, "An object-based methodology for knowledge representation in sgml," in *Proceedings of the Ninth IEEE International Conference on Tools With Artificial Intelligence*, pp. 304–311, IEEE Computer Society, (Los Alamitos, CA), 1997.
9. R. C. Schrag and M. Soukup, "Proposal for high performance knowledge bases project - challenge problem development and eavaluation management," December 1996. Information Extraction and Transport, Inc.
10. Mosaic, National Center for Supercomputing Applications `http://www.ncsa.uiuc.edu/SDG/Software/Mosaic`.
11. Navigator, Netscape Communications Corporation `http://www.netscape.com`.