An Object-Based Methodology for Knowledge Representation in SGML

Robert. L. Kelsey
Los Alamos National Laboratory *
New Mexico State University

Roger. T. Hartley
New Mexico State University

Robert. B. Webster
Los Alamos National Laboratory

Abstract

An object-based methodology for knowledge representation and its Standard Generalized Markup Language (SGML) implementation is presented. The methodology includes class, perspective, domain, and event constructs for representing knowledge within an object paradigm. The perspective construct allows for representation of knowledge from multiple and varying viewpoints. The event construct allows actual use of knowledge to be represented. The SGML implementation of the methodology facilitates usability, structured, yet flexible knowledge design, and sharing and re-use of knowledge class libraries.

1 Introduction

An object-based methodology for knowledge representation uses objects of classes as its fundamental representational building block. The overall concept of using objects as a building block is not unlike what is done in object-oriented analysis (OOA) and design (OOD) [3] [2]. Each of these decomposes knowledge into manageable units called objects. It is natural to decompose knowledge into objects and objects are modular enough to allow for changes and extensions within the representation.

This methodology is a design methodology (not a representation system) for designing domains of knowledge. Knowledge representation systems often skip the step of designing the knowledge that the system represents. As a result, knowledge representation systems are difficult to use. It may be possible that knowledge designed using this methodology can later be put into existing knowledge representation systems.

This methodology includes the use of perspectives and events. Perspectives allow for the representation of knowledge from multiple viewpoints. This can increase the detail of the knowledge and put the knowledge in additional context. Events allow for the representation of actual use of the knowledge along with the knowledge itself.

The use of Standard Generalized Markup Language (SGML) to implement the methodology adds structure, but maintains flexibility. Also, SGML facilitates the sharing and re-use of knowledge through it portability and constructs. A less obvious, but important aspect of the SGML implementation is its added usability. A growing familiarity of markup languages helps make the SGML implementation of the methodology more usable. The following sections describe the methodology (briefly) and the SGML implementation. More detail about the methodology can be found in [7].

2 Methodology background

2.1 Introduction

Knowledge is represented in the methodology using three different constructs: the class, the domain, and the event. A class is a template or prototype that describes an object or collection of objects. Once a class is defined it is an available resource for creating a domain of knowledge. A domain is a specific focus of knowledge. Classes are instantiated for use in a domain. Instantiation causes specific instances or objects of those classes to become available for use. This is why the domain can be focused, because only the classes of interest or pertinence need be instantiated. An event is a construct for representing examples of actual use of the knowledge in a domain.

*The Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; therefore, the Laboratory as an institution does not endorse the viewpoint of a publication or guarantee its technical correctness.

2.2 Class

There are two types of classes possible in the methodology, agent and non-agent. Although the types are not formally designated in notation, the distinction is useful for determining interactions of and between objects of classes. Others (particularly Schank [11]) have used the same distinction in representation, calling agent objects actors. Agent objects of a class act on or initiate actions on other objects of both agent and non-agent classes. Non-agent objects of a class are essentially inanimate objects.

A class consists of three parts: attributes, methods, and perspectives. An attribute is a property or characteristic that helps describe an object of a class. An attribute has a name or identifier and a value. It may also have a list of potential or limiting values. The value of an attribute is assigned at the time an object is instantiated from a class and can be changed by methods of the object.

A method determines the behavior of an object of a class. A method is an allowed operation on an object. Objects interact with one another through methods and limiting the methods available can constrain interaction between objects. A method has a name or identifier, a list of arguments (if any), and the operation content. The operation content is how a method goes about doing what it does. This may be represented in a variety of ways including rules, pseudocode, or even a program. A method is not available for use until an object is instantiated from a class.

A perspective defines how or in what way an object of a class may be viewed. Some attributes may be important to one perspective while not important to another perspective. Thus, a particular perspective is defined by grouping the attributes and methods in a class that are pertinent to the perspective. Similar to a method, a perspective on an object is defined in that object's class and limits or constrains the available views. Each attribute and method in a class has a list of perspectives with which it is associated.

The name or identifier and the definition of a perspective go in the class that possesses that perspective or view. There is also a list of classes that are a part of this class's perspective. A perspective can be considered to be how one object of a class views the other objects in a domain.

2.3 Domain

A domain is where objects of classes are instantiated for use. At instantiation, objects are created and named and values within the parts of an object are assigned. Only the knowledge necessary for a particular application need be defined in a domain. A domain has a name and a list of objects contained within the domain. An object (within the domain definition) has a name or identifier, the name of the class from which it was instantiated, and if it is an agent object, the perspective with which it is viewing the rest of the domain.

2.4 Event

A domain of objects can contain or represent knowledge and maybe even how the knowledge is used, but it cannot necessarily represent actual examples of use. An event is specifically designed for representing an example use of knowledge. An event has a name or identifier and consists of four parts: a before state, a method, an after state, and an agent. A state is a snapshot of an object containing the values of an object's attributes. A method is the name of the method causing the change in state from before to after. An agent is the name of the agent object instigating the method to change the state.

A single event represents only two objects, the agent object and the object being acted upon by the agent object. Additionally, only the object attributes that change values from before to after state are listed within the states. This is in keeping with the spirit of only representing what is necessary.

3 SGML background

3.1 Introduction

The Standard Generalized Markup Language (SGML) is an ISO standard for the description of documents [6]. This standard describes technology for facilitating text interchange in documents [12], 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 multi-media. 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.

3.2 SGML components

There are three components or parts to an SGML document [1]. 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. For example, a memo document may contain content like who the memo is to, who the memo is from, and a message. Markup for this document will label or tag those corresponding parts as the to, the from, and the message.

3.3 SGML DTD

There are four constructs used in creating a DTD: the element, the attribute, the entity, and the comment [8]. Note that the attribute construct should not be confused with the attributes of a class discussed previously. The DTD is a grammar that defines the parts to a document, their structure, and their notation in terms of the four constructs mentioned previously.

An element construct can be thought of as a container [8]. Any information that is to be contained must have an element defined to hold the information. For example, the message portion of a memo document will have a message element. There may also be to and from elements. These elements hold and contain the content of the message portion, to portion, and from portion (respectively) of the memo document.

An attribute construct is attached to an element construct and is used to provide further detail about the element. For example, the message element may have an attribute called security type. The security type attribute designates whether the message is unclassified, sensitive, or classified. An attribute construct may be thought of as a modifier of the element construct.

An entity construct is used to contain portions of document content that are repeated throughout the document. By defining an entity, content is defined once, but can be used many times. In this way, an entity is like a macro or an include file. If the content changes, it must only be changed in one place. A comment construct is used to express comments or notes that are not part of the document content and not meant for the end user to see.

4 Implementing the methodology

Implementing the object-based methodology in SGML involves creating a DTD that maps the methodology constructs and their parts into SGML constructs. The following sections describe the SGML constructs for the three methodology constructs and their parts. In the following sections note that when the word attribute is used it refers to the attribute part of the class construct in the methodology. An SGML attribute for an SGML element will be referred to as SGML attribute or SGML attribute construct. Names of the created SGML constructs (and example values) are shown in boldfaced sans serif type. Names and values in the examples in figures are shown in proper SGML notation. The (uncommented) DTD created for the object-based methodology is in Appendix A.

4.1 The class element

The class construct becomes an element called **class**. The three parts to a **class** (attributes, methods, and perspectives) each become elements as well. Contained within the **class** element are the elements called **attribute**, **method**, and **perspective**. The **attribute** element may contain an element called **potvals**. The **potvals** element contains a list of potential values that an **attribute** may possess. The **class** and **attribute** elements each have an SGML attribute construct attached to them called **name**. In both, the **name** holds the name of the **class** and the name of the **attribute**, respectively.

The **method** element may contain the elements called **args** and **content**. The **args** element contains a list of arguments (if any) to the **method**. (Any elements that contain lists may use an additional markup called **li** to designate an individual list item.) The **content** element contains the operation content of the **method**. The **method** element also has an SGML attribute construct attached to it called **name** which holds the name of the **method**. The **method** and **attribute** elements each have an additional SGML attribute construct attached to them called **perspect**. In both, **perspect** holds a list of perspectives with which the **method** and **attribute** (respectively) are associated and/or grouped.

The **perspective** element may contain an element called **def**. The **def** element contains the definition of the **perspective**. There are two SGML attribute

constructs attached to the **perspective** element called **name** and **classes**. **Name** holds the name of the **perspective** and **classes** holds a list of classes that are observed within this **perspective** or view.

4.2 The domain element

The domain construct becomes an element called **domain**. The **domain** element may contain an element called **instance**. **Instance** refers to an object being instantiated from a **class**. The **domain** element has an SGML attribute construct attached to it called **name** which holds the name of the domain. The **instance** element has three SGML attribute constructs attached to it called **classname**, **name**, and **perspect**. **Classname** holds the name of the **class** from which the **instance** or object is being instantiated. **Name** holds the name of the **instance** or object. **Perspect** holds the name of the **perspective** this **instance** or object uses to view the rest of the domain. Within the **instance** element is an element called **attval**. The **attval** element contains the actual value of an **attribute** of the **instance** or object. The **attval** element has an SGML attribute construct attached to it called **name**. **Name** holds the name of the particular **attribute** in the **class** to which this value is assigned.

4.3 The event element

The event construct becomes an element called **event**. The **event** element may contain four elements called **befstate**, **meth**, **aftstate**, and **agent**. The **befstate** element contains information about the before state. The **meth** element contains the name of the **method** causing the change in state. The **aftstate** element contains information about the after state. The **agent** element contains the name of the agent object that instigated the **method** to change the state. The **event** element has an SGML attribute construct attached to it called **name** which holds the name of the **event**.

5 An AI example

To illustrate how the SGML implementation is used and what it looks like, a classic problem from artificial intelligence (AI) will be presented. The blocks world is one of the classic problems in AI [9] [10] [13] and comes from a collection of problems known as microworlds [10]. Microworld problems have limited knowledge domains and appear to need intelligence to solve.

The blocks world contains a table, blocks, and a robot arm. The robot arm manipulates the blocks into different arrangements on the table, such as stacks of blocks. The blocks world is a good example to illustrate this methodology and the SGML implementation because it is a narrow domain of knowledge, but contains enough to exercise most of the methodology's constructs.

Note that the figures illustrating each of the constructs use the notation and markup as defined by the SGML DTD created for the methodology. If this research was about document construction and not knowledge representation, then the examples shown in the figures would be known as document instances. The markup language has an opening and closing tag for each element defined in the DTD. An opening tag is delimited by the < and > symbols while a closing tag is delimited by the </and > symbols. The name of the element defined in the DTD appears between the tag delimiters. If the element has an SGML attribute construct attached to it, then the value for the SGML attribute appears (in quotes) in the opening tag of the element.

5.1 Class definitions

The blocks world has three classes of objects which are table, block, and robot arm. The **class** for table is the simplest because it contains no attributes or methods. In this domain of knowledge, the table is merely a physical object where the blocks may be located. Defining more than this in the **class table** is extraneous and irrelevant. The **class robot-arm** has one **attribute** named **holding**. The **holding attribute** contains the knowledge of whether the robot arm is holding a block and if so, which block. The **holding attribute** may have potential values of **NULL** (meaning not holding a block) or a block's identification (**ID**).

The **class robot-arm** is an agent **class**, so it contains a **perspective** named **stacking** whose view of the domain includes the **classes** of **table** and **block**. The definition of the **perspective** is **the typical view of the blocks world**. Figure 1 shows the **class** definitions for **table** and **robot-arm**.

```
<class name="table"></class>


<class name="robot-arm">
<attribute name="holding">
<potvals>
<li>NULL
<li>an ID
</potvals>
</attribute>


<perspective name="stacking"   classes="table block">
<def>
the typical view of the blocks world
</def>
</perspective>
</class>
```

Figure 1: Class definition for classes table and robot-arm.

The **class block** has three attributes named **ID**, **location**, and **covered**. All three attributes are grouped into a perspective (**perspect**) named **stacking**. The **ID** is a block's identification. The **location** is a block's current location and has potential values of **table** or an **ID** (meaning on top of a block). The **attribute covered** indicates whether a block is on top of this block and if so, which block (is on top) and has potential values of **NULL** (meaning not covered) or an **ID** of a block. Figure 2 shows the definition for the **class block**.

```
<class name="block">
<attribute name="ID"   perspect="stacking"></attribute>
<attribute name="location"   perspect="stacking">
<potvals>
<li>table
<li>an ID
</potvals>
</attribute>
<attribute name="covered"   perspect="stacking">
<potvals>
<li>NULL
<li>an ID
</potvals>
</attribute>
<method name="pickup"   perspect="stacking">
<content>
IF robot-arm.holding is NULL THEN
  IF covered is NULL THEN
    location := robot-arm
    robot-arm.holding := ID
  ELSE
    error "block covered"
ELSE
  error "arm is holding"
</content>
</method>
<method name="putdown"   perspect="stacking">
<args>
<li>locale
</args>
<content>
IF robot-arm.holding not NULL THEN
  IF locale = an ID THEN
    IF locale.location = an ID THEN
      error "stack too high"
    ELSE
      location := locale
      robot-arm.holding := NULL
  ELSE
    location := locale
    robot-arm.holding := NULL
ELSE
  error "not holding block"
</content>
</method>
</class>
```

Figure 2: Class definition for the class block.

The **class block** has two methods which are **pickup** and **putdown**. Both of these methods are grouped into a perspective (**perspect**) named **stacking**. The **method putdown** has an argument named **locale** which is the location of where the robot arm will put down a block. The operation **content** of each of the methods is shown in pseudocode. Although these methods sound more like methods for the robot arm, they are methods in the **class block** because these are operations on a block. Actually, putting the methods in the **class robot-arm** was tried. The result was that it did not seem to matter. The methods can be put in either **class**, but it is important to select and use one convention consistently. One possible advantage to having these methods in the **class block** is that they act as a constraint. These are the only operations allowed on a block. The **content** of the methods is also shown in Figure 2.

5.2 Domain and event definitions

An example **domain** for the blocks world contains a **table** object named **table1**, a **robot-arm** object named **robot-arm1**, and four **block** objects named **blockA**, **blockB**, **blockC**, and **blockD**. The name of the **domain** is **blocks-world1**. The object **robot-arm1** is instantiated with the value **NULL** for its **attribute holding**. **Robot-arm1** is an agent object and views the rest of the **domain** from the **stacking** perspective (**perspect**). The four **block** objects are each instantiated with the values of their respective names for the **attribute ID** and the values **table1** and **NULL** for the attributes **location** and **covered**, respectively. Figure 3 shows the definition of the **domain blocks-world1**.

An example **event** named **event1** illustrates **blockA** being picked up by **robot-arm1**. The **befstate** shows **blockA**'s **location** to be on **table1** and **robot-arm1** is not holding a block. The **meth pickup** of **blockA** is executed. The **aftstate** shows **blockA**'s **location** to be **robot-arm1** and **robot-arm1** is holding **blockA**. The **agent** instigating the method is **robot-arm1**. Figure 3 shows the definition of the **event event1**.

5.3 Another perspective

The SGML implementation of the methodology allows for easy extension to the existing knowledge. Suppose that in addition to the **stacking perspective** of the blocks world there is a perspective to teach a child to spell. In this new perspective the blocks may be alphabet blocks and words are spelled by placing the blocks next to each other. This new perspective is named **spelling** and will add attributes and methods to the existing **class block**. Also, the labels of which attributes belong to which perspectives will change.

In the **class block** there are two new attributes called **rightside** and **leftside** which may have potential values of **NULL** (meaning no block present to the particular side) or a block's identification (**ID** of block on the particular side). These two attributes belong within the perspective (**perspect**) **spelling**. The **attribute ID** and the **method pickup** are added to the **perspect spelling** and continue to also remain in the

```
<domain name="blocks-world1">
<instance classname="table"  name="table1">
</instance>
<instance classname="robot-arm"  name="robot-arm1"  perspect="stacking">
<attval name="holding">NULL</attval>
</instance>
<instance classname="block"  name="blockA">
<attval name="ID">A</attval>
<attval name="location">table1</attval>
<attval name="covered">NULL</attval>
</instance>
<instance classname="block"  name="blockB">
<attval name="ID">B</attval>
<attval name="location">table1</attval>
<attval name="covered">NULL</attval>
</instance>
<instance classname="block"  name="blockC">
<attval name="ID">C</attval>
<attval name="location">table1</attval>
<attval name="covered">NULL</attval>
</instance>
<instance classname="block"  name="blockD">
<attval name="ID">D</attval>
<attval name="location">table1</attval>
<attval name="covered">NULL</attval>
</instance>
</domain>


<event name="event1">
<befstate>
blockA.location = table1
robot-arm1.holding = NULL
</befstate>
<meth>
blockA->pickup
</meth>
<aftstate>
blockA.location = robot-arm1
robot-arm1.holding = blockA
</aftstate>
<agent>
robot-arm1
</agent>
</event>
```

Figure 3: Definitions for the domain blocks-world1 and the event event1.

perspect stacking. There is one new method named **putbeside** which may have no arguments (put on the table) or two arguments, an **ID** and a **side** (of a block). The **method putbeside** is grouped within the **perspect spelling** and allows blocks to be placed beside one another. Note that knowledge to actually spell is not represented. Figure 4 shows the definition of **class block** (method content not shown) with the added **perspect spelling**. Only the changes and extensions are shown, otherwise the definition is the same as in Figure 2.

In the **class robot-arm** is the added **perspective spelling** with its definition (**def**) **setting blocks side by side for spelling words**. The list of classes within this perspective are **table** and **block**. The changes and extensions to the **class robot-arm** are also shown in Figure 4.

```
<class name="block">
<attribute name="ID"  perspect="stacking spelling">
</attribute>
...
<attribute name="rightside"  perspect="spelling">
<potvals>
<li>NULL
<li>an ID
</potvals>
</attribute>
<attribute name="leftside"  perspect="spelling">
<potvals>
<li>NULL
<li>an ID
</potvals>
</attribute>
<method name="pickup"  perspect="stacking spelling">
...
</method>
...
<method name="putbeside"  perspect="spelling">
<args>
<li>NULL
<li>an ID
<li>side
</args>
</method>
</class>


<class name="robot-arm">
...
<perspective name="spelling"  classes="table block">
<def>
setting blocks side by side for spelling words
</def>
</perspective>
</class>
```

Figure 4: Definitions of class block and robot-arm with the added perspective spelling.

5.4 Potential reasoning

Lines of reasoning are possible using perspectives and events. Different perspectives on the same object and the same perspective on different objects may suggest useful comparisons and contrasts. Gentner [5] describes comparisons of objects and domains based on literal similarity, analogy, and abstraction which are derived from the mappings of objects, object attributes, and relations between objects. Consider the example of the blocks world with the two different perspectives **stacking** and **spelling**. These are two different views of the same sets of objects. It is fairly simple to observe that a stack of blocks laid down will look like a line of blocks side by side, but can a reasoning mechanism make this observation? Perspectives may help facilitate this type of reasoning.

Events may be useful for another potential line of reasoning. Consider the **event event1** from Figure 3. Suppose the method (**meth**) was left blank. Given the before state (**befstate**), after state (**aftstate**), and class definitions (with method contents) it may be possible to infer the method (**meth**) executed to change the state. Actually, given any two of the three

(**befstate**, **meth**, **aftstate**), the third may be inferred with the help of the class definitions and their method contents. It may be possible to make other inferences by including the **agent** with the other **event** elements.

Although not discussed in this paper, the methodology and its SGML implementation have the ability to create classes using the concepts of inheritance and aggregation/disaggregation. More detail about these concepts is discussed in [7]. Using inheritance and aggregation/disaggregation relationships can allow for additional lines of reasoning.

6 Other features

There are some other features to the methodology that are made available by the SGML implementation. One of these features is the ability to share and re-use knowledge. Part of this is facilitated by the notion of containing knowledge in classes and part of it is facilitated by SGML. Class libraries of common knowledge may be created and distributed in an SGML form. A class can be defined within an entity construct in a DTD. This will allow the class to be included and used in other knowledge domains by referencing the defined entity. That fact that SGML is very portable will also contribute to the sharing of knowledge.

Another feature is the potential ease of use of creating representations using SGML. The increasing popularity of the World Wide Web (WWW) has increased the use and understanding of markup and tagging languages. Using SGML to implement the methodology exploits this growing familiarity with markup languages. In fact, in order to make the creation of knowledge (in SGML) easier and quicker for testing the methodology, an Emacs mode for the Free Software Foundation's GNU Emacs [4] was developed. The mode has control-key sequences defined for all the methodology constructs and their associated tags. Additionally, a knowledge domain and the DTD for the methodology can be checked (within Emacs) against a validating SGML parser. In the future, code will be added to the Emacs mode to reason across some of the knowledge.

7 Conclusions

The object-based methodology and its SGML implementation complement each other. Together they provide a structured, yet flexible means of representing knowledge. The available constructs allow knowledge to be limited and focused to a specific problem or application area. Only the pertinent knowledge need be represented. Should the focus later change, the existing knowledge can be easily changed and/or extended.

The perspective and event constructs add detail to the knowledge being represented. Perspectives allow for the representation of multiple and varying viewpoints. Events allow for the representation of actual knowledge use. Both these constructs may facilitate new lines of reasoning over the knowledge domain.

Additionally, the SGML implementation helps make the knowledge more usable and portable. SGML documents can be interchanged independent of hardware and software and there is no proprietary format [12]. Also, by using the SGML entity construct, class libraries of common knowledge can be created, shared, and re-used.

More research is yet to be done on exercising the methodology and its SGML implementation on more complex knowledge domains. The limits of the methodology in what types of knowledge it may represent must be identified. Also, it should be compared to other representation schemes and systems. More research must also be done to further understand and implement the potential reasoning mechanisms in the methodology including reasoning with perspectives and events.

References

[1] Liroa Alschuler. *ABCD...SGML A User's Guide To Structured Information.* International Thomson Computer Press, Boston, MA, 1995.

[2] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley Publishing Company, Menlo Park, CA, second edition, 1994.

[3] Peter Coad and Edward Yourdon. *Object-Oriented Design.* Yourdon Press, Englewood Cliffs, NJ, 1991.

[4] Free Software Foundation, Boston, MA. *GNU Emacs Lisp Reference Manual*, 2.4 edition.

[5] Dedre Gentner. *Readings in Cognitive Science A Perspective from Psychology and Artificial Intelligence*, chapter 3.2, Structure-Mapping: A Theoretical Framework for Analogy, pages 303–310. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[6] International Organization for Standardization, Geneva. *ISO 8879:1986 Information processing - Text and office systems - Standard Generalized Markup Language (SGML)*, October 1986.

[7] Robert L. Kelsey, Roger T. Hartley, and Robert B. Webster. An object-based methodology for knowledge representation. Technical Report LAUR 97-1598, Los Alamos National Laboratory, Los Alamos, NM, 1997.

[8] Eve Maler and Jeanne El Andaloussi. *Developing SGML DTDs From Text to Model to Markup.* Prentice Hall PTR, Upper Saddle River, NJ, 1996.

[9] Elaine Rich and Kevin Knight. *Artificial Intelligence.* McGraw-Hill, Inc., New York, NY, second edition, 1991.

[10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[11] Roger C. Schank and Robert P. Abelson. *Scripts Plans Goals and Understanding An Inquiry Into Human Knowledge Structures.* Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ, 1977.

[12] Eric van Herwijnen. *Practical SGML.* Kluwer Academic Publishers, Boston, MA, second edition, 1994.

[13] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley Publishing Company, Reading, MA, second edition, 1984.

A DTD for the methodology

```
<!ELEMENT kr - - (class*, domain*, event*)>
<!ELEMENT class - - (attribute*, method*, perspective*)>
<!ELEMENT domain - - (instance*)>
<!ELEMENT attribute - - (potvals? | instance?)>
<!ELEMENT method - - (args?, content?)>
<!ELEMENT args - - (li+)>
<!ELEMENT content - - (#PCDATA)>
<!ELEMENT perspective - - (def)>
<!ELEMENT instance - - (attval*)>
<!ELEMENT potvals - - (li+)>
<!ELEMENT attval - - (#PCDATA)>
<!ELEMENT def - - (#PCDATA)>
<!ELEMENT li - o (#PCDATA)>
<!ELEMENT event - - (befstate?, meth?, aftstate?, agent?)>
<!ELEMENT befstate - - (#PCDATA)>
<!ELEMENT meth - - (#PCDATA)>
<!ELEMENT aftstate - - (#PCDATA)>
<!ELEMENT agent - - (#PCDATA)>
<!ATTLIST class name NAME #REQUIRED
          inherit NAME #IMPLIED>
<!ATTLIST attribute name NAME #REQUIRED
          perspect NAMES #IMPLIED>
<!ATTLIST method name NAME #REQUIRED
          perspect NAMES #IMPLIED>
<!ATTLIST perspective name NAME #REQUIRED
          classes NAMES #IMPLIED>
<!ATTLIST instance classname NAME #REQUIRED
          name NAME #REQUIRED
          perspect NAMES #IMPLIED>
<!ATTLIST attval name NAME #REQUIRED>
<!ATTLIST domain name NAME #REQUIRED>
<!ATTLIST event name NAME #REQUIRED>
<!ATTLIST kr name NAME #REQUIRED>
```