A Hierarchical Systems Knowledge Representation Framework

by

Igor Andrade Sylvester

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

Author .... [illegible]
Department of Electrical Engineering and Computer Science
September, 2007

Certified by .... [illegible]
Daniel Hastings
Professor
Thesis Supervisor

Accepted by .... [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Hierarchical Systems Knowledge Representation Framework

by

Igor Andrade Sylvester

Submitted to the Department of Electrical Engineering and Computer Science
on September, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We present the design and implementation of a framework for storing and analysing knowledge about engineering systems. The hierarchical entity-relation-attribute model is useful for large data sets, in which it can abstract details so that human users are able to reason about the data. The time-series extension to the model abstracts temporal details. Finally, the implementation of the model includes an execution engine that can simulate the model in one time-slice or as a function of time.

Thesis Supervisor: Daniel Hastings
Title: Professor

Contents

Chapter 1

Introduction

This thesis project consists of two parts. First, we develop a knowledge framework to represent, analyze and persist engineering systems. Second, we describe the implementation of Frog; a software application for handling knowledge frameworks.

The knowledge framework developed in this thesis extends the classical entity-relation model by introducing the concept of object-oriented inheritance from computer science. We introduce inheritance as a means of abstraction to understand large engineering systems. In addition, our framework is capable of incorporating, querying, and simulating temporal data. The temporal nature of the framework is limited to variables–and not the topology–of engineering systems. We hope future work in this area can provide an implementation to time-varying system topologies.

The main design goal for Frog is to speed up data-acquisition and simplify the visualization of systems knowledge. The application is based on the concept of Design Structure Matrices ("DSM") and the system decomposition framework of [1]. The key design requirements include a collaborative environment with support for concurrent data access, a persistent data storage system and an intuitive graphical user interface ("GUI"). The collaborative requirement of the application fits naturally the framework of a 2-tier (client/server) system. The server provides data persistence and basic business logic. The client interacts with the server and exposes the GUI to the end user.

I have also decided to consider a visual attachment metaphor for data exploration.

A visual component will perform a small set of functions. Upon the completion of each, a message will be generated and passed to the components parent. In general, the parent will be the main application. However, designers of visual components will be able to invoke other visual components from their own, leveraging the visualization and the functionality of the child component, without having to implement it themselves. It is hoped that this metaphor will yield a consistent, more intuitive interaction.

1.1 Background

A DSM is a tool that can be used to model complex systems. Stripped to its essence, a DSM is a compact form for representing the dependencies in a system. Each of the entities in the system takes a position on the diagonal of the matrix. The dependencies between entities are indicated by entries in off-diagonal cells. It is an approach that can be used for analyzing systems as well as planning and managing projects.

As a planning tool, it can be used to identify when many entities have a single dependency. If that dependency is one employee, perhaps the work needs to be divided to mitigate risk. If that entity is a physical component, then perhaps many resources can be dedicated to it so that it is built quickly and that it is reliably robust.

From a computer science perspective, a DSM is the adjacency matrix for a directed graph with colored edges. The nodes and the edges are called the entities of the system. The entities posess a set of attributes (name/value pairs) which provide them with identity. Tacitly, a DSM represents one or more individual's decomposition of a system and its internal dependencies. Any organization that wanted to use a DSM for decision-making would want to be able to verify the decomposition. They would want to be able to answer the question of "why". Why do we think that Entity A depends on Entities B, C and D? The system should be able to allow a user to associate an entity or relation with the documents that are its source – preferably, the pertinent snippets of those documents.

Different people think in different ways. Some analysts will prefer to interact with an adjacency matrix while others will want a graph that they can explore, node by node, while still others will just want a table view. This preference for presentation notwithstanding, multiple users will need to be able to collaborate on the same DSM; changes made by one user need to be propagated to the other users. These requirements imply the presentation of the system should be able to render the model in multiple fashions and that those renderings should be able to respond to data changes in a timely fashion.

The originial entity-relation model was presented in [4]. The author claims that the model posseses most of the advantages of the network model, the relational model and the entity set model. The concept of an entity is defined as a "thing" which can be distinclty identified; and a relationship is defined as an associated among entities.

The semantic web is another popular knowledge framework. It is a natural language extension to the world wide web. At its core, the semantic web is based on the resource description framework ("RDF"). The RDF model is based upon the idea of making statements about resources in the form of subject-predicate-object expressions, called triples.

The entity-relation and semantic web models are different manifestations of a directed graph. In the entity-relation model, the nodes of the graph represent the identities while the edges represent the relations. Similarly in the semantic web model, subjects and objects correspond to nodes and predicates correspond to edges.

While there are software packages that provide DSM capabilities, they do not solve the functional requirements in a sufficiently general fashion to allow for a natural decomposition. Rather, they tend to focus on one aspect of the process or one type of system. Those that are more general achieve their generality by erecting substantial barriers to entry; they have complicated user interfaces.

The existing software models' DSMs lack expressiveness. Two examples of this terseness are in the representations of relationships and of temporally varying entities. These packages' exclusively focus on mere dependence – they do not allow the users to distinguish between the types of dependencies that two entities may have on one another. They don't address time varying entities, relationships or attributes at all.

Finally, the existing tools tend to be difficult to use: they have too many options and often require class time to learn how to use effectively. Moreover, they don't optimize any particular stage of DSM creation or use, making it difficult to manage DSMs when the number of entities becomes large.

1.2 Modeling Languages

System modeling languages such as the Entity-Relationship model (ER model), the Unified Modeling Language (UML) and the Object-Process methodology (OPM) provide syntactic and semantic definitons of building blocks to represent real-world systems [9].

The ER model leverages a graphical language that describes the static structural relationships between entities. It provides an intuitive and topological view of systems. The ER model adopts the view that the real world consists of entities and relationships. It incorporates semantic information about the real world and makes use of set and relational theory to achieve data-independence [4].

UML is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system [10]. It is a general-purpose modeling language that can be used will all major object and component methods, and that can be applied to all application domains and implementation platforms. UML was originally designed for software systems but it has been introduced in other fields. Thus, the description of UML has grown to adapt to other environments.

OPM is a visual modeling language with a single diagrammatic view and a small set of symbols. It offers a superior alternative to UML. The UML community has regarded that the complexity of UML hinders system modeling. OPM was created as simpler alternative to UML while the community works on redefining UML.

1.3 Engineering Systems Frameworks

On top of modeling languages, modeling frameworks have been introduced to better understand system level interactions. A modeling framwork provides lexical tools to system designers for constructiong descriptions of complex systems. Some modeling frameworks depend on modeling languages to describe systems. Bartolomei [2] gives a summary of the following modeling framewoks: Quality Functional Deployment (QFD), Design Structure Matrix (DSM), Unified Program Planning, Axiomatic Design Framework, and Complex Large Integrated Open System (CLIOS).

Quality Functional Deployment (QFD) accumulates information about the social and technological domains and the inter-relations between domains, and intra-relations withnin classes of information. A QFD framework modeling task starts with the customer needs to engineering characteristics, interactions between engineering characteristics, and target values for the engineering characteristics [5]. System analysts use the framework to prioritize customer needs, understand and envaluate the system parameters and their performance implications. Critical limitations to the QFD framework include a homogeneous set of stakeholders and not including social interactions.

Unified Program Planning (UPP) is a more general framwork than QFD. While QFD was primarily designed for product development, UPP is used for policy analysis methodology for non-nengineering systems. The UPP metholody provides a structural approach for representing traceability within complex systems.

The Axiomatic Design Framework represents system design consisting of four domains: the customer, functional, physical, and process domains. The Complex Large Integrated Open Systems framework provides a methology and abstractions to describe complex social and technological systems.

Design Structure Matrices provide a way to describe inter- and intra-domain interactions in five domains. The domains include "the goals domain the product (or service, or result) system; the process system (and the work done to get the product system); the system organizing the people into departments, teams, groups, etc.; the systems of tools, information technology solutions, and equipment they use to do the work; and the system of goals, objectives, requirements, and constraints pertaining to all the systems." [6]. The graphical nature of the ER model can support various structural relationships. The ER model is used in software engineering to represent inheritance diagrams for Object-Oriented design.

1.4 Temporal Entity-Relation Models

Several temporal extenstions to the ER model have been developed. Gregersen and Jensen [8] provide a survey of temportal Entity-Relationship models. The authors claim that ten temporally enhanced ER models had been reported in the research literature until the time of publication in 1999. Summaries are provided for the Temporal Entity-Relationship Model, the Relationships, Attributes, Keys, and Entities Model, the Model for Objects with Temporal Attributes and Relationships, the Temporal EER Model, the Semantic Temporal EER Model, the Entity-Relation-Time Model, the Temporal ER Model, the TempEER Model, the TempRT Model, and TERC+.

The survey concludes that the basic ER model in itself does not provide adequate support for elegantly and concisely capturing temporal aspects of data. The temporal extensions to the ER model take fundamentally different approaches. One approach is to introduce new syntactic notation to replace common patterns that arise in ER models with a temporal aspect. A different approach is to introduce new semantics to the ER model constructs, making them temporal.

1.5 Synopsis

Chapter 2 introduces the concept of engineering systems and develops the Hierarchical Entity-Relation-Attribute model.

Chapter 3 describes the computer implementation of the model. The implementation includes a programmable database storage engine. An API is provided for technical users.

Chapter 4 presents the graphical interface that leverages the programmable database. This interface is provided for non-technical users.

Chapter 5 concludes the thesis and presents future extensions and improvements to the model, database and user interface.

Chapter 2

Engineering Systems

For the purpose of this thesis, an engineering system is an integrated composite of people, products, and processes that provide a capability to satisfy a stated need or objective. This thesis is not concerned about the theory or applications of engineering systems. Instead, this thesis develops a computational framework to represent and manipulate engineering systems. In other words, we present a knowledge framework for engineering systems.

Nevertheless, in order to design the framework, it is necessary to understand the basics of engineering systems. Bartolomei gives the following axioms for an engineering system:

1. It is composed of interacting technical and social/organizational components that exists within an economic, legal and political context.

2. It is a system of purpose that is defined and valued by human entities.

3. It is large in scale or consists of many interacting parts that exhibit non-trivial behavior.

4. It is complex or exhibit structural, behavioral, and/or interface complexity

5. It evolves with varying rates of change.

6. It exhibits emergent properties.

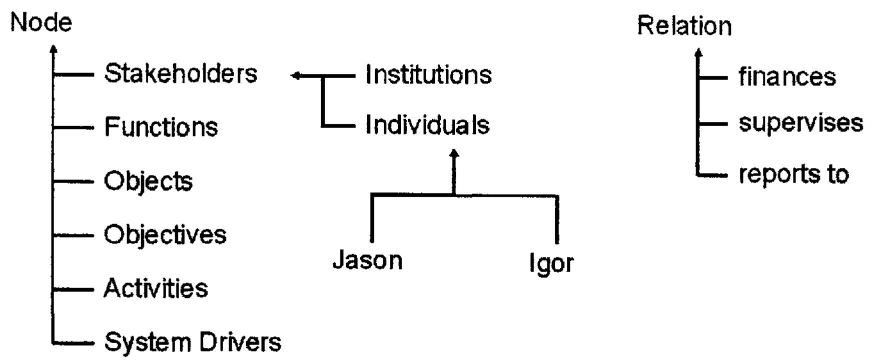


Figure 2-1: The hierarchical part of a trivial engineering system. The system ontology describes the hierarchy of nodes and edges (or relations). Here, Node is the parent of the first-level ontological sub-systems (Stakeholders, Functions, etc). In turn, Igor and Jason are children of Individuals; this, in turn, is a child of Stakeholders. Similatly, there are three relations that are children to "Relation."

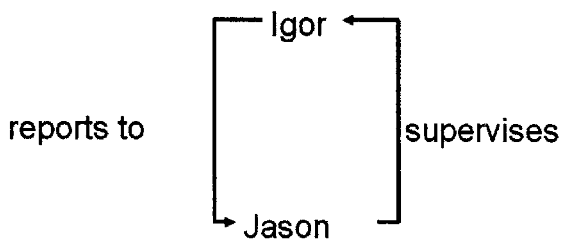


Figure 2-2: The network (or graph) part of an engineering system. There are two nodes: "Igor" and "Jason", and two edges: "reports" to and "supervises". The network topology states that Igor reports to Jason and Jason supervises Igor.

7. It is an open system.

The knowledge framework developed in this thesis incorporates each of these axioms. In order to gain intuition, let's consider a trivial instance of an engineering system. We pospone the formal definition of the model to a later section. The framework is composed of two structures: a hierarchical model (see Figure 2) and a topological graph (see Figure 2). In addition, attributes maintain information about the nodes and relations in the graph (see Figure 2).

This chapter explains design concepts that were used to develop the knowledge framework. After motivating our design decisions, each functional part of the model is described. The following chapter continues with the computer implementation of the model.

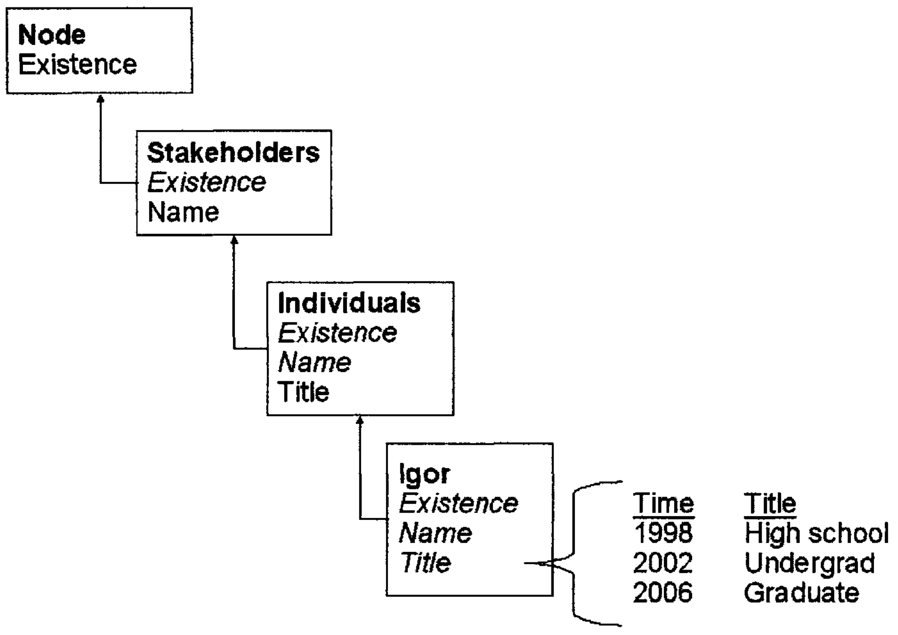


Figure 2-3: The hierarchy for a node and the inheritance of attributes. The root node "Node" has the attribute "Existance." In turn, "Stakeholders" has the attribute "Name" and "Existance", which is inherited from its parent. Similarly, "Individuals" introduces the attribute "Title" and inherits two attributes. Finally, "Igor" only inherits attributes and specifies a value for the attribute "Title." The value of this attribute is a set of time range/value pairs, which give the knowledge model its temporal nature.

2.1 Meta-object Protocols

A Meta-object protocol is an specification of how objects behave in a system and how they are be manipulated. For example, in object-oriented computer programming languages, such as Java and C++ for example, the meta-object protocol is built into the compiler. In these languages, the protocol specifies class inheritance, polymorphism, field access control, and method forwarding. Some languages support dynamic meta-object protocols, i.e. the protocol is available during runtime. For example, Java provides an introspection library that is able to create and manipulate classes and invoke methods programatically. Similarly, Common Lisp's meta-object protocol is available during runtime; classes are created during program evaluation. In both of these cases, classes are first-class objects, i.e. classes are instances of a meta-class. Hence, the name meta-object protocol.

Since engineering systems can be thought of in terms of classes and objects, a meta-object protocol is an obvious computer implementation. The remainder of this chapter develops a meta-object protocol specifically suited for engineering systems.

2.2 Classic and Prototypical Inheritance

In object-oriented computer programming languages, inheritance usually refers to class inheritance. In single-inheritance languages such as Java, class inheritance defines a class tree structure, where the edges denote an inherits-from relation. On the other hand, in multi-inheritance languages such as C++, class inheritance defines an acyclic class graph. Classic inheritance is useful when there are many objects from a given class. However, in the case of a singleton class–which has only one object instance–it is unconvenient to define a class for the sole purpose of instanciating a single object.

Prototypical inheritance solves the singleton class problem. In a prototypical inheritance object-oriented programming-language, objects themselves inherit from other objects. Thus, objects serve as prototype for new objects. The dynamicallytyped programming language Javascript supports a prototypical inheritance. Since object instantiation is coupled with class definition, the meta-object protocol must be accessible during runtime. For example, it is common for Javascript programs to add member fields and methods dynamically. As a result, the source code for Javascript programs that rely heavily on singletons is much simpler than the equivalent source code for other class-inheritance languages.

2.3 The Hierarchical Entity-Relation Model

In this section, we formally define the knowledge framework model. The structure should abstract complexity by introducing hierarchy betweeen nodes and edges in the DSM. In short, the systems knowledge framework consists of a tree and a directed graph. The tree encodes the hierarchical abstractions and the graph represents the topology of the DSM. Alternatively, the graph encodes qualitative information about the engineering system while quantitative information is kept by attributes. Attributes are name/value pairs that belong to entities of the DSM. Later in the chapter, we specify functionality to manipulate the quantitative portion of the DSM by evaluating mathematical expressions held by entity attributes.

The meta-object protocol for the hierarchical entity-relation model is summarized in Figures 2.3 and 2.3. Note that the model is general in the sense that both nodes and edges follow classic inheritance. In the implementation of Frog, we chose to use prototypical inheritance for nodes. In other words, nodes and node-types have a one-to-one relationship. This makes the code and user interphase easier to work with.

2.4 The Time-series Attribute Model

In the previous section, we defined a model that is able to represent sentences composed of nouns and verbs. This model is very simple, however. For example, nouns and verbs are central to the English language. However, English would not be useful

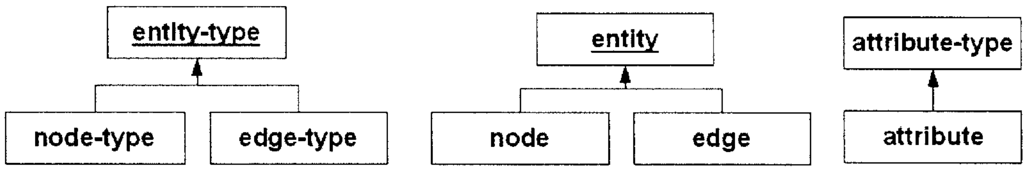


Figure 2-4: The class model for a classic-hierarchical knowledge representation framework. The classes node-type and edge-type inherit from edge-type and represent the schematics for building entities. This, in turn, is sub-classes by node and edge. Finally, attribute-types serves as the schematic for attributes.

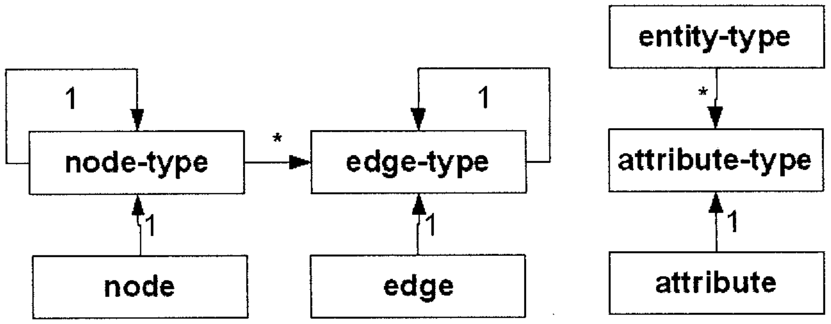


Figure 2-5: The object model for a classic-hierarchical knowledge representation framework. The template of nodes and edges is derived from their repestive node-type and edge-type. Each of these, in turn, have a parent, which gives the framework its hierarchical structure. Similarly, attributes derive their structure from a attribute-type. Entity-types keep a collection of attribute-types which restricts the attributes for entities.

if it were limited to only nouns and verbs. We would like to add expressiveness to the language–and that is what adjectives do. Adjectives modify the character of nouns and verbs. So, we extend the entity-relation model by introducing the equivalent of adjectives: attributes.

Formally, each node and edge, i.e. entity, has a set of attributes. In turn, each attribute consists of a time/value pair.

2.5 Naming Convention

We define a naming convention for nodes, edges and attributes. First, let's consider an example system. We describe the formal naming convention afterwards. Consider a trivial system with two nodes with attributes and an edge. The nodes are named

$$Objects.A$$
$$Objects.B$$

The objects' canonical names are "A" and "B" and their mutual parent is the node named "Objects". Node A has an attribute color:

$$Objects.A : color$$

From this name, we can immediately read the entity's name. Similarly, the value of color at times 1 and 2 are referenced with the names:

$$Objects.A : color.1$$
$$Objects.A : color.2$$

Relations have a similar naming convention. Since relations have a source and target nodes, they form part of the name:

$$Objects.A > related > Objects.B$$

Similary, relation's attributes are referenced like

$$Objects.A > related > Objects.B : strength$$

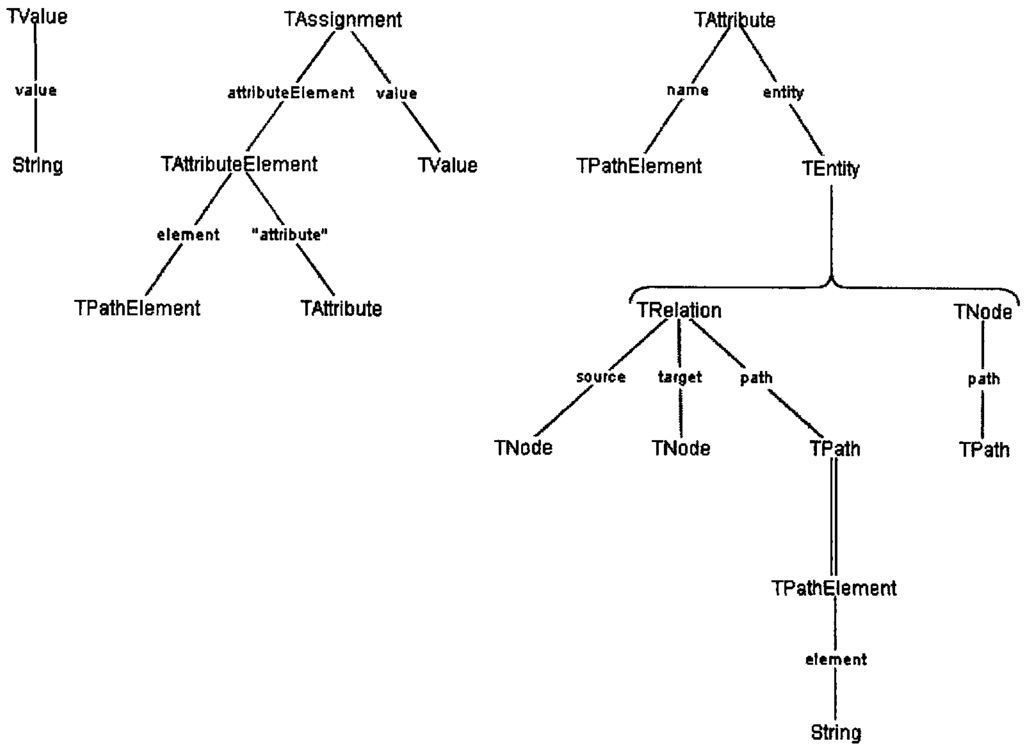


Figure 2-6: The token class structure for the implementation of the grammar for describing knowledge atoms. Nodes are identified by a path. A path is a sequence of strings. A relation consists of its name and two nodes. An attribute is composed of an entity and a path element, i.e., a string. An assignment represents the value assignment of an attribute element. A value is an infix mathematical expression which may have references to other attributes.

and the respective time elements have names like

$$Objects.A > related > Objects.B : strength.1$$

The naming convention presented here is convenient for quickly referencing and creating database objects. See Figure 2.5 shows the abstract parse tree for the naming grammar. In addition, the engineering system may be thought of as a set of names. Each name corresponds to a knowledge atom of the system. In particular, relation names map to sentences in English, in which the source node corresponds to the subject, the relation corrresponds to the verb and the target node corresponds to the verb object. So, the knowledge framework is just a collection of statements about the system.

2.6 The Executable Model

The Executable model refers to the simulatable portion of the knowledge framework. We developed a simulation engine only for the quantitative part of the model, i.e. the time-series attribute model. The development of a simulation engine for the qualitative part of the framework, i.e. for the topology of nodes and edges, is recommended for future work.

The simulation engine evaluates attribute values as functions of other attribute values in the system. Functional expressions are evaluated independently for each time step. Thus, the simulation is static. The source code for the simulator is included in the Appendix. Future work can extend this paradigm to mix-in different time steps.

The representation of formulas is trivial after considering our naming convention from earlier in the chapter. For illustration purposes, consider a node "X" with two attributes: "a" and "b". We can initialize the attributes by stating

$$X.a.1 = 1$$
$$X.a.2 = 2$$
$$X.b = X.a + 1$$

The first two assignments set the values of 1 and 2 to time elements 1 and 2 of attribute a. The last assignment states that all time elements of b are evaluated to 1 plus the value of the corresponding element in attribute a. So, $X.b.1 = 2$ and $X.b.2 = 3$. Circular references are not allowed in this simulation scheme. In other words, $X.b = X.b + 1$ is an invalid expression.

2.7 Typical Data Queries

Besides looking up nodes and edges, there are interesting queries on the knowledge framework. For example, given a node, we can find all the nodes that are at a certain distance (measured in number of edges) away. Below is LISP code for this functon, which we call n-reachable.

```
;;;
;;; n-reachable finds all nodes
;;; that are d edges away from node n.
;;;
(defun n-reachable (d n)
  (if (eq n 0)
      ;; node is 0 edges away from itself
      (list n)
      ;; bag holds the nodes
      (let ((bag))
        ;; for all nodes x that are connected to n
        (dolist (x (nodes n))
          ;; add all the nodes that are reachable
          ;; from node x with d-1 edges
          (setf bag (append bag
                    (next-node (n-reachable (- d 1) x)))))
        bag)))
```

2.8 Database Querying and Searching

The database provides functionality to dynamically retrieve entity objects. This is critical to develop a robust user interface and it also prrovides the foundations of a scripting language for manipulating the model.

The core search functionality is to convert the name of an entity, given as a string, into the database representation of the entity. For example, a query for "Objects.1" returns the entity object

```
(query ''Objects.1'') ==> <node ''1''>
```

Similarly, we can query for edges, attributes and attribute-elements:

```
(s "Objects.1>b>Objects.2")
```

```
(s "Objects.1>b>Objects.2:strength")
(s "Objects.1>b>Objects.2:strength.1")
```

Before giving the definition of the `search` function, we need to define a couple of helper functions. The function `select` takes a parsed node or edge-type name and returns the object, or creates it, if it does not previously exists. The arguments to `select` are:

1. `class` A class object; either node or edge-type. It specifies what type of object to select.

2. `pnode` The parsed name of a node or edge-type as a list of strings. The original name is broken at individual node boundaries. For example, "Objects.1" turns into ("Objects" "1").

3. `parent` An optional argument that specifies the parent of the new object, if one is created.

```
(defun select (class pnode &optional parent)
  ;; Prepare three variables for later use.
  (let* ((child-name (first pnode))
 (test-child (find-if (lambda (x) (equal child-name (name x)))
                      (children parent)))
         (child (if test-child
    test-child
             (make-instance class :name child-name :parent parent))))
   ;; If pnode still contains names
   (if (cdr pnode)
       ;; recurse
       (select class (cdr pnode) child)
       ;; otherwise, the object is in child.
child)))
```

The function select-edge finds, or creates, and returns the edge of type et between nodes out and in This function is the equivalent of select for nodes. It deals wit the extra subtlies of classical inheritance.

```
(defun select-edge (out et in)
  (let ((test-edge (find-if (lambda (e) (eq et (edge-type e)))
                            (directed-edges out in))))
       (if test-edge
           test-edge
           (make-instance 'edge :edge-type et :in-node in :out-node out))))
```

The function query uses select and select-edge for low-level querying of objects. Thus, query needs to parse object names and delegate querying to the appropriate function. For convenience, we do not present the definition of the scanner (lexer). We need only to know that the scanner makes available the function parse-with-lexer which tokenizes a string using the grammar given in *spidr-parser*.

```
(defun query (text)
  (let ((object (parse-with-lexer (spidr-lexer text) *spidr-parser*)))
    (case (first object)
        ('node (select 'node (second object) (root-node spidr)))
('edge (let ((out (select 'node (second cmd) (root-node spidr)))
     (in (select 'node (fourth cmd) (root-node spidr)))
     (edge-type (select 'edge-type (third cmd) (edge-type (root-edge spidr)))))
 (when (and out in)
   (select-edge out edge-type in)))))))
```

Finally, we present the definition of the fuzzy search function, search. This function implements predictive text entry. For example, searching for "Obj" returns the node "Objects".

```
(defun search (q &key (type :nodes) (spidr *spidr*))
  (cond ((eql :nodes type)
 (multiple-value-bind (matched all parent extra)
     (match-re "(.*)[.]+([^.]*)" q)
   (let ((child (or extra q)))
     (remove-if
      (lambda (x) (or (> (length child) (length (name x)))
      (not (equal child (subseq (name x) 0 (length child))))))
      (let ((p (if matched
   (query parent)
   (root-node spidr))))
(when p (children p)))))))
(t nil)))
```

Chapter 3

Frog: The Framework Implementation

Using the knowledge model developed in the previous chapter, we are ready to describe Frog, the software implementation for the knowledge framework. Frog's main purpose is to serve as a knowledge repository. Additionally, Frog facilitates the management, sharing, and visualization of knowledge. To this end, the software toolkit requires:

- **The Entity Panel** is intended to be the common search mechanism and/or data entry for links and nodes. The entity box will be a common interface in basic search and coding entry. In the short term, the entity box is used in the qualitative coding tool for coding text. In the long term, the entity box will be used in the ESM and Table View for rapid search or new node creation.

- **The Search Box** provides functionality to quickly find specific entities and attributes within the DSM.

- **The Attribute Panel** contains the information that describes each entity. The entity pane contains

- **The Table View** is an interactive table that allows the users to see all of the entities simultaneously. The user has the option to view nodes, relations, or all entities at once. Each class of node exists in designated columns.

- **The ESM (Matrix) View** is an interactive window of the ESM with nodes along the diagonal and relations in off diagonal cells.
- **The Query Panel** provides an interphase to execute non-trivial operations of the DSM.
- **The Coding Panel** is used to document the sources for the entities.

Data integrity is important for any database design. Traditional Relational Database Management Systems ("RDMS") achieve data integrity by requiring ACID-compliance. Similarly, Frog aims at implementing ACID-compliance in its design. The requirements for ACID-compliance are atomicity, consistency, isolation and durability. We explain how Frog implements ACID-compliance in its storange engine and graphical interface.

3.1 Global Architecture

Frog has a client/server design. Many clients connect to a single server. The server stores (or persists) the knowledge database and provides updates to the clients. The server application was written in the Franz [7] implementation of Common Lisp. The client is implemented in Java SE 1.5. We chose to use Lisp to write the server because Lisp provides maximum flexibility during the prototyping phase of development. On the other hand, Java was used for the client because it may be deployed in multiple platforms and it can easily be delivered over the Internet using Java WebStart.

Alternatives to using AllegroCache include SQL and BerkeleyDB [3] databases. We decided that the SQL model is not appropriate for our needs. Specifically, the relational SQL model does not map cleanly with hierarchical structures. Storing trees in SQL databases is a common problem with multiple solutions. There are two major approaches: the adjacency list model, and the nested set model. We attempted to build a prototype implementation using the adjacency list model using te Hibernate persistance layer. We found that a SQL database provides many features that are not needed. Specifically, Alternatively, we believe that BerkeleyDB provides features in Java that are similar to the features provided by AllegroCache. Therefore, BerkeleyDB is a more appropriate database implementation.

The AllegroCache persistance layer has comparable load and capacity limitations compared to other commercial database implementations such as MySQL.

Independently using the best implementation tools for the client and server does not imply that the global architecture is optimal. We found several impedance mismatches at the interface between the client and server, i.e. between Java and Common Lisp, respectively. The main reason causing this impedance mismatch is data persistance.

3.2 Data Persistance Objects

The knowledge framework meta-object protocol is represented using a set of Common Lisp Object System ("CLOS") class objects. The objects are persisted using the prevalence framework AllegroCache. Unlike a relational database, where data storage is based on records, a prevalence system is based on objects. Thus, a prevalence system is better suited for persisting graph-like and hierarchical structures than a relational database.

Nodes follow the prototype inheritance model (see Figure 3.2) and relations follow the classic inheritance model (see Figure 3.2).

The implementation of inheritance of attributes for nodes is redundant. After a node is created and associated with a parent node, attributes from the parent node are replicated in the newly created node. This process is repeated for all the ancestors of the node. Whenever an attribute is added to a node, the attribute is replicated into all of the node's descendants. Similarly, the delition of an attribute is replicated as well.

Inheritance of attributes for relations works similarly as for nodes. After a relation is instanciated and associated with a relation type, its attribute set is populated with

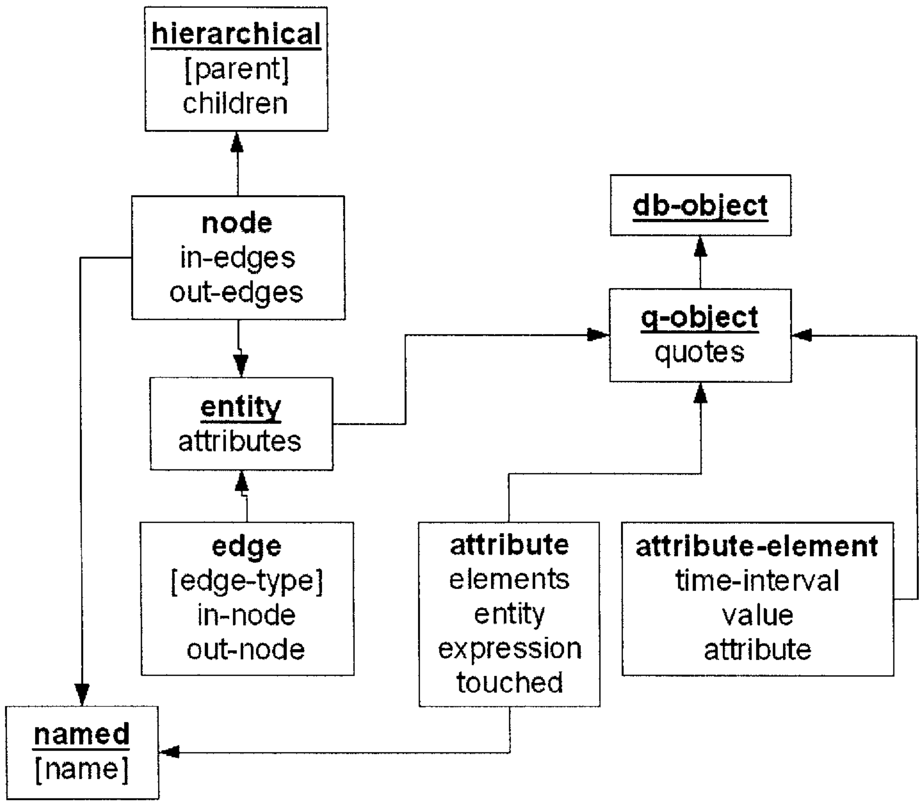


Figure 3-1: Class structure for the prototype-hierarchy portion of the knowledge representation framework. The concrete classes are node, edge, attribute and attribute-element. The abstract classes are hierarchical, entity, named, db-object, and q-object. Field members expressed in square brackets represent redundant information solely for purposes of indexing. For example, the hierarchical class has two fields: parent and children. The field parent holds the id number of the parent and children holds a list of id numbers of its children. The parent field may be derived from children. Thus, the parent field is redundant; it's purpose is for efficiency only.

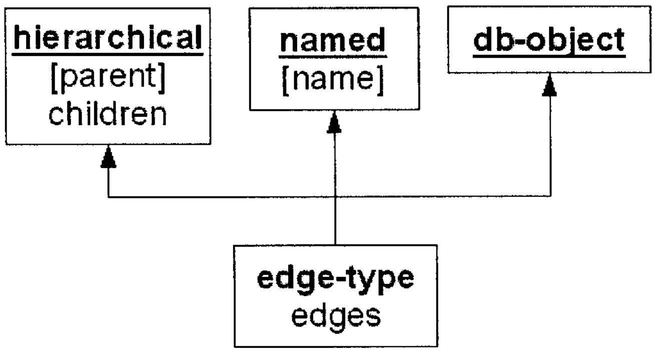


Figure 3-2: Class structure for the classic-hierarchy portion of the knowledge representation framework . The abstract classes presented here are the same as in Figure 3.2. The only concrete class is edge-type.

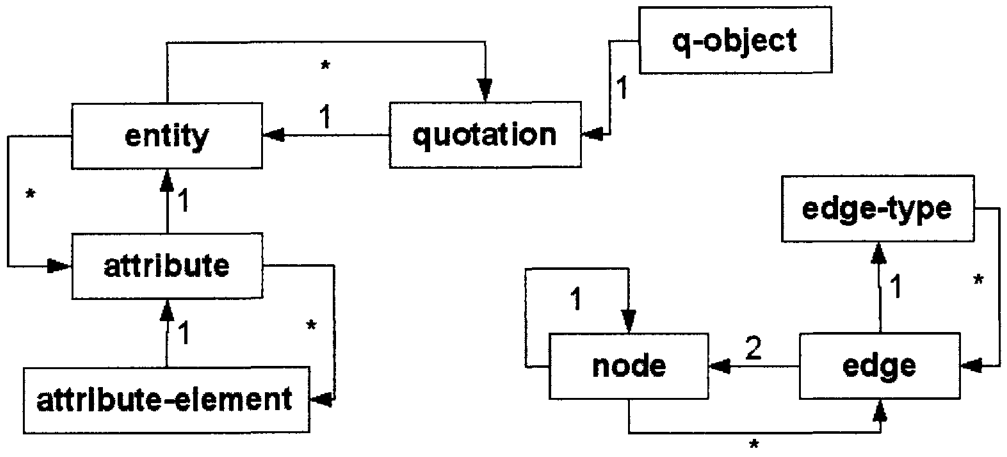


Figure 3-3: The runtime object model for the hybrid knowledge representation framework.

fresh copies of attributes from the relations of the same type. Thus, all instances of relations of the same type share similar sets of attributes. The attributes have different elements and values in each relation, however.

The runtime object model of the framework is shown in Figure 3.2.

3.3 Attribute Inheritance

This section explains the implementation of attribute inheritance. Inheritance is triggered after entity and attribute creation. When an entity (node or edge) is created, we need to assign it new attributes which depend on its context. Nodes inherit attributes from their parents while edges inherit attributes from their type.

Below is the code for a couple of helper functions that we'll later use.

```
;;; has-attribute returns true if, and only if, entity has an
;;;     attribute named name.
(defun has-attribute (entity name)
  (find name (attributes entity) :key #'name :test #'equal))
```

;;; copy-attribute constructs a a copy of attribute and assigns ;;; it to entity.

```
(defun copy-attribute (attribute entity)
  (let ((new-attr (make-instance 'attribute
                                 :entity entity
                                 :name (name attribute)
                                 :expression (expression attribute))))
    (dolist (e (elements attribute))
      (make-instance 'attribute-element :attribute new-attr
                     :time-interval (cons (car (time-interval e))
                                          (cdr (time-interval e))
                     :value (value e)))))
```

Four methods are responsoble for attribute inheritance. The method `inherit` is called after an entity is created. Similarly, the method `inherit-into` is called after an attribute is created.

```
;;; Inherit the attributes from node's parent.
(defmethod inherit ((self node))
  ;; Copy attributes from parent
  (let ((parent (parent self)))
    (when parent
      (dolist (attr (attributes parent))
              (copy-attribute attr self)))))

;;; Inherit the attributes from the edge's type
;;; Effectively, copy all the attributes from any
;;; pre-existing instance of edge-type
(defmethod inherit ((self edge))
  (let ((siblings (remove self (edges (edge-type self)))))
    (when siblings
      (let ((sibling (car siblings)))
        (dolist (attr (attributes sibling))
          (copy-attribute attr self))))))
```

For a new attribute in a node, the following function makes a copy of the attribute for each of the node's descendants.

```
;;; Inherit-into ensures that the newly created attribute
;;; is propagated into all of the node's descendents.
(defmethod inherit-into ((self attribute) (n node))
  ;; copy attribute to child nodes
  (let ((name (name self)))
    (dolist (child (children n))
      (unless (has-attribute child name)
        (copy-attribute self child)))))
```

For a new attribute in an edge, the following function makes a copy of the attribute for each of the instances of the edge-type.

```
;;; Inherit-into ensures that the newly created attribute
;;; is propagated into all of the instances of the edge's edge-type
(defmethod inherit-into ((self attribute) (e edge))
  ;; copy attribute to sibling edges
  (let ((siblings (remove self (edges (edge-type e))))
        (name (name self)))
    (dolist (sibling siblings)
      (unless (has-attribute sibling name)
        (copy-attribute self sibling)))))
```

3.4 Documents and Quotations

Frog stores plain-text files ("documents") in its database. Documents are ment to beassociated with nodes, relations, and attributes. These associations are encapsulated and persisted as quotations. Quotations relate a continuous region of text in a document and a quotable object.

Figure 3.4 shows the class structure for documents and quotations.

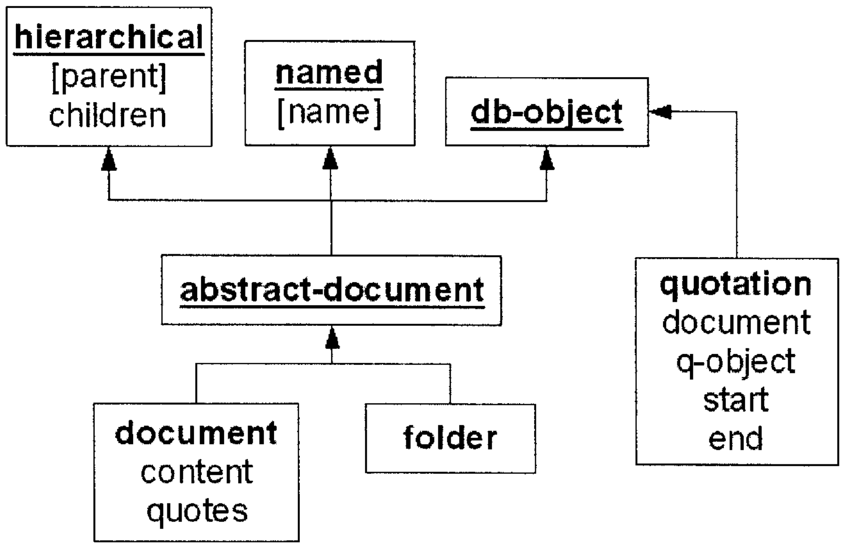


Figure 3-4: Class structure for documents and quotations.

3.5 Application Programming Interface

The server encapsulates all the functionaity described in the previous sections and provides an XML remote program procedure (XML-RPC) interface to the client.

Every object in the database (node, edge, etc.) has a unique identifier that is initialized automatically and never changes. The database can store many engineering systems.

- **spidr.systems** Returns a list of the names of the engineering systems stored in the database.
- **spidr.parse** Takes a name for an object (node, edge, attribute, attribute element) and returns its identifier, if it exists. Otherwise, it creates the object and returns its identifier. The source code for this function is listed in the appendix.
- **spidr.query** This function is a soft version of `spidr.parse`. It returns all the identifiers of the objects that have names . The source code for this function is listed in the appendix.
- **spidr.get** Given a valid identifier, this method returns relevant information about the node, edge, attribute, attribute element, document, folder, system or

quotation.

- **spidr.rootNode** Given a system identifier, this method returns the identifier of the root node.
- **spidr.makeSystem** Makes and initializes a new system. The system is populated with a root node and a root edge type.
- **spidr.makeNode** Created a new node. The parent node is set to the root node by default.
- **spidr.makeEdge** Creates an edge between two nodes. If the edge type does not exist, then one is created.
- **spidr.makeAttribute** Creates an attribute and binds it to the entity. The attribute is propagated immediately following the rules of hierarchy.
- **spidr.makeAttributeElement** Adds a time element to the attribute. The value is initially empty.
- **spidr.makeFolder** Creates a new folder for organizing documents.
- **spidr.makeDocument** Creates a new document inside the specified folder.
- **spidr.makeQuotation** Creates a new quotation.
- **spidr.remove** Given a valid object identifier, this method removes the object from the system.
- **spidr.removeSystem** Gives a valid system identifier, this method removes the system and all its objects from the database.
- **document.setContents** After creating a document, this convenience method loads its contents.
- **attributeElement.setTimeInterval** Given an identifier for an attribute element, this method sets its time interval. The time format is left unspecified. The end-user has the choice of using any time convention suitable.

- **attributeElement.setValue** Sets the value of an attribute element. Values are strings which can be valid mathematical expression (including object names). Non-mathematical expressions are ignored--and treated as non-existent--for the simulation engine.

- **node.setPositionOf** This method sets the order of a child node in its parent list of children. This ordering is used solely for convenience in the front-end.

Chapter 4

Graphical User Interface Implementation

The GUI front-end to Frog is implemented in Java 1.6. It communicates with the database through the XML-RPC interphase described in the previous chapter.

The GUI implementation follows the Model View Controller design pattern. The model and controller classes live in the *model* package. The View classes live in the *gui* package.

A description of the packages follows below.

- **model.dao** Holds the data access objects, which are responsible for loading and representing database objects.

- **model.commands** Atomic commands that operate on the database. This set of classes follows the command design pattern.

- **gui.tree** Contains FrogTree and supporting classes, which are leveraged throughout the entire application.

- **gui.attributeManager** The attribute manager panel displays information about nodes such as their relations and attributes.

- **gui.entityManager.table** The table view of the model is a set of customized tree components that are optimized for fast data entry.

- **gui.entityManager.esm** The ESM view of the model is a customized hybrid tree-table component that attempts to mimic the ajacency matrix of the graph.
- **gui.documentCoder** A text-file views that allows coding regions of documents.
- **gui.documentManager** A simple filesystem for storing documents.

4.1 Model Objects

The Data Access Objects ("DAO") are responsible for storing and accessing a partial local copy of the database in the client.

- **Client** represents a connection to the Frog database. It is a wrapper for the XML-RPC client.
- **System** represents an engineering system. It provides methods for creating nodes, edges, documents and quotations.
- **AbstractDocument** parent class of Document and Folder.
- **Document** represents a text document. Methods are available for adding and retrieving quotations.
- **Folder** represents a folder that can hold documents and other folders.
- **DatabaseObject** is the abstract class from which all persistent classes inherit from.
- **Entity** is the abstract class from which Node and Edge inherit from. It provides functionality to add and retrieve attributes.
- **QuotableObject** this abstract class is the parent class of all the classes that can be associated with a document though a quotation.

- **StateObject** the abstract class responsible for making API calls using an instance of the Client class. Given an object id, this class loads the information associated with such object.

- **Node,Edge,Attribute,AttributeElement,Quotation** these classes represent the basic building blocks of a knowledge base.

- **TimeInterval** represents an iterval of time. It is currently implemented as a pair of strings that specify starting and ending times in an unspecified date format.

4.2 Command Objects

The operational part of Frog follows the command pattern. Operations in Frog are abstracted as objects and stored in the *model.Commands* package. There are two reasons for using the command pattern. First, to facilitate future implementation of an extended database persistance engine and to implement a robust MVC design.

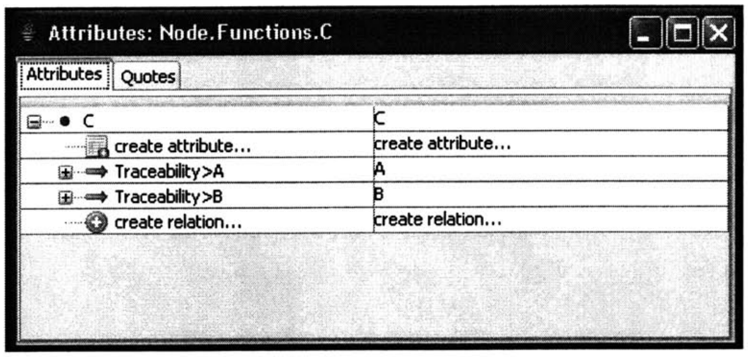
The command objects can be used to implement a native prevalence persistence engine and break away from proprietary implementations such as AllegroCache. A native prevalence system can be used to implement undo and collaboration capabilities. In this context, each command object maps to an atomic operation.

The command pattern is a critical component for the MVC design. The command objects are used as signals when the database object models change. Viewer classes can suscribe to model objects and receive notifications about updates. The command object encapsulate the information concerning the model updates.

Frog currently implements the following operations:

- **QueryCommand** A non-destructive operation. Returns a list of database objects that have a name that match the query text.

- **SetDocumentContentsCommand** This operation sets the contents of a document. It is meant to be used only once immediately after creating a document.



C	C
create attribute...	create attribute...
Traceability>A	A
Traceability>B	B
create relation...	create relation...

Figure 4-1: The Inspection pane for a node. Nodes in the graph contain a set of attributes and relations.

- **SetTimeIntervalCommand** Sets the time interval of an attribute element.

- **SetElementValueCommand** Sets the value of an attribute element. The value can either be numerical or an attribute-annotated mathematical expression.

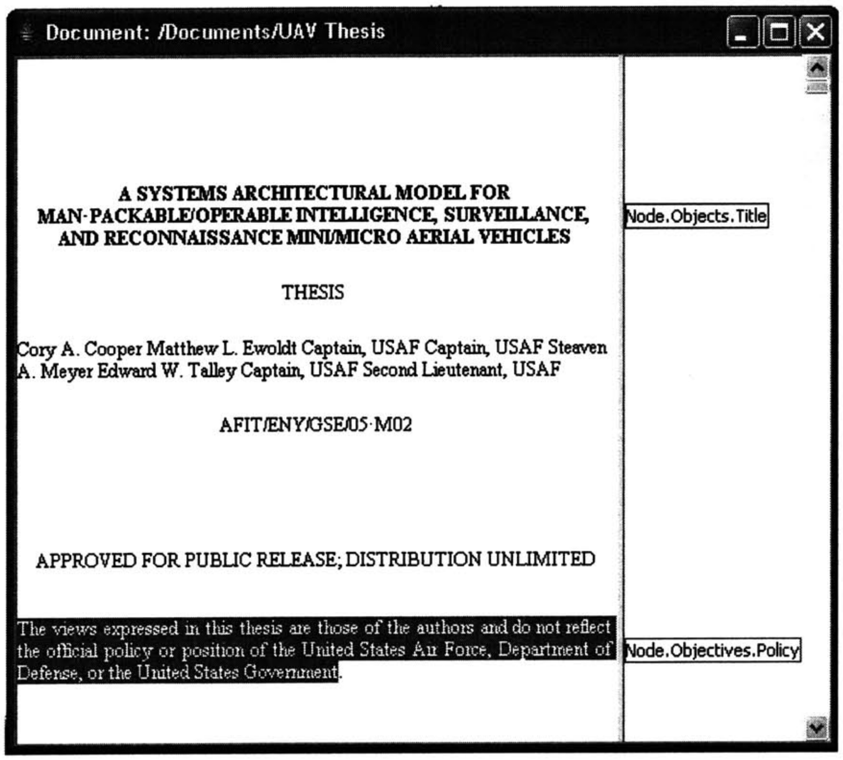
- **SetPositionOfCommand** This operation exists only for the convenience of the front-end GUI. The list of children of a node persists order. This operation sets the order of a node in the children list.

- **Make**-command classes represent the creation of database objects. There are make-commands classes for Nodes, Edges, Attribute, Attribute Elements, Documents, Quotations and Systems. Each of these classes encapulate the necessary information to create the respective objects.

- **Remove**-command complement the make-command classes and encapsulate the necessary information to remove objects from the system.

4.3 Inspection Pane

The Inspection Pane is designed to provide detail descriptions of entities. Figure 4.3 shows the inspection pane for a node.


Document: /Documents/UAV Thesis

A SYSTEMS ARCHITECTURAL MODEL FOR
MAN-PACKABLE/OPERABLE INTELLIGENCE, SURVEILLANCE,
AND RECONNAISSANCE MINI/MICRO AERIAL VEHICLES

THESIS

Cory A. Cooper Matthew L. Ewoldt Captain, USAF Captain, USAF Steaven
A. Meyer Edward W. Talley Captain, USAF Second Lieutenant, USAF

AFIT/ENY/GSE/05-M02

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

Figure 4-2: The Qualitative coder. A document is coded by assigning quotable objects (nodes, edges, attributes and elements) of the graph to segments of text.

4.4 Document Coder

The document coder is a tool to document the source of knowledge in the model. Entities can be associated with portions of text of a text file. The quotations are displayed on the right side of the pane. Figure 4.4 shows a document along with codes within the document coder.

4.5 Entity Manager

The Entity Manager is the principal window of user interaction in Frog. There are two visual representations: a table view and a matrix view. The table view is optimized for rapid data entry. In contrast, the matrix view provides a graphical representation of the adjacency matrix of the entity-relation graph.

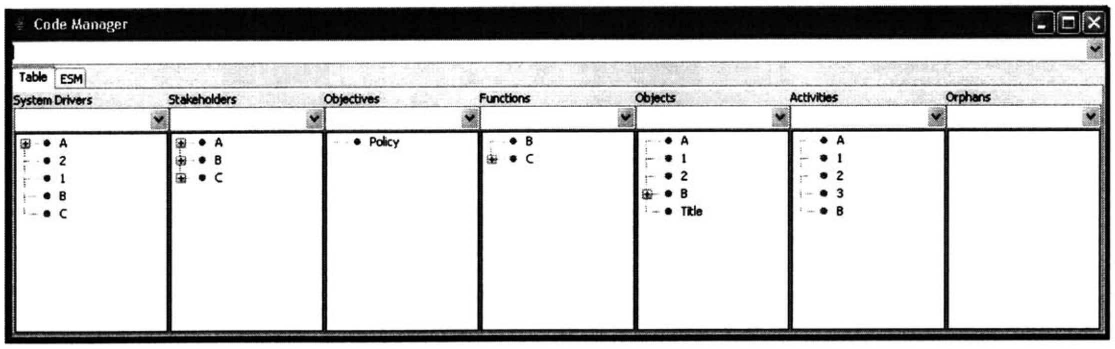


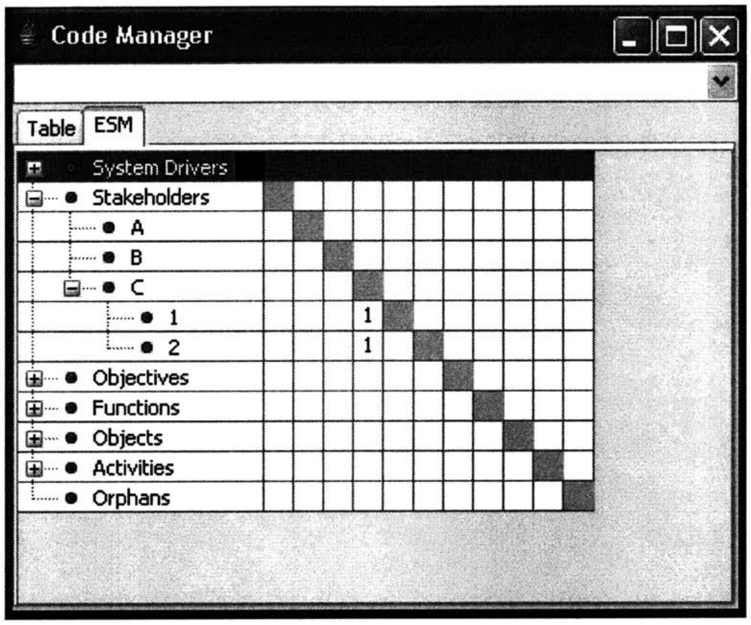
Figure 4-3: The Table view of the entity manager. The nodes are grouped by their ontological character.

Table View

The Table View is an interactive table that allows the users to see all of the entities simultaneously. The user has the option to view nodes, relations, or all entities at once. Each class of node exists in designated columns. Figure 4.5 shows the table view populated with nodes and edges.

ESM View

The ESM View is a custom-made Java Swing component that renders an interactive adjacency matrix of the database object graph. Figure 4.5



System Drivers											
Stakeholders	▨										
A		▨									
B			▨								
C				▨							
1				1	▨						
2				1		▨					
Objectives							▨				
Functions								▨			
Objects									▨		
Activities										▨	
Orphans											▨

Figure 4-4: The ESM view of the entity manager. On the left, the ontology tree shows the nodes as rows. On the right, the cells of the adjacency matrix show the number of directed edges between nodes.

Chapter 5

Conclusion

We developed a hierarchical entity-relation model and implemented a user interphase for analysing and hanlding engineering systems. The tool developed for this project is capable of handling qualitative and quantitative information. Qualitative information is encoded in the entity-relation topology while quantitative information is encoded in the attributes of nodes and edges.

5.1 Future work and Improvements

There are many ways in which the hierarchical entity-relation model and its implementation can be improved and extended. In this section we summarize extensions to the knowledge model and improvements to the server and client architectures.

Database Design

The Common Lisp implementation of the database offers flexibility during prototyping. Unfortunately since Lisp is not a popular language, the maintenance and future development of the codebase will be restricted by the skill set of developers. A practical solution is to port the current implementation into Java. Java is far more popular than Lisp and has a larger set of libraries available. In addition, Java compilers are open-source and free of charge. In contrast, the Franz Common Lisp compiler needs a license to use. We do not expect any complications in porting the codebase to Java because the class structure is defined and the Lisp implementation does not use features that are not already available in Java.

The best alternative to the AllegroCache persistance layer is the Java Edition of BerkeleyDB provided by Oracle. BerkeleyDB offers a light-weight persistance layer for Java objects. The schema of the database is defined through Jave class annotations, which define key and index fields. BerkeleyDB does not provide a query language. Instead, all access to the database is through lookup of primary and seconday keys. Since the Lisp implementation of the knowledge store performs lookups only through primary keys, BerkeleyDB is an excellent alternative to AllegroCache.

Scripting

The GUI front-end offers a descriptive graphical representation of the data. The user interacts though keyboard and mouse inputs. However, more advanced users can benefit from macro or scripting functionality. Effectively, scripting exposes part of the back-end API to the end-user. Applications that leverage scripting such as Excel and Emacs have proven to be more extensible and useful.

In the current server implementation, scripting functionality can easily be added by defining Lisp functions or macros that compile a scripting language into code that manipulates AllegroCache persistant objects. This thesis has already given a few building blocks for this design. We have defined a larguage for naming nodes, edges and attributes. A basic scripting language can completed by adding logic and algorithmic structures such as for loops, assignments, variable scoping, etc. The Lisp implementation of this language would not need to implement all of these features because it can leverage the meta-linguistic capabilities of Lisp. In contract, a Java implementation of the scripting language would not be able to leverage this functionality because Java does not provide a meta-circular evaluator. Instead, we recommend the use of open-source implementation of scripting languages such as Python in Java.

Future developers of Frog should consider using Jython [11]–a popular Java implementation of Python. The developers would only need to provide hooks into Jython to bring the database back-end functionality into Python.

Appendix A

Selected Snippets of Source Code

A.1 CLOS Class Object Definitions

```
(defclass* hierarchical ()
  ((parent :index :any) children))
(defclass* named ()
  ((name :index :any)))
(defclass* spidr (named)
  (root-node root-edge root-folder
  (sobjects :initform (make-instance 'ac-set))))
(defclass* db-object ())
(defclass* q-object (db-object)
  (quotes :initform nil))
(defclass* entity (q-object)
  (attributes))
(defclass* node (entity hierarchical named)
  (in-edges out-edges))
(defclass* edge-type (db-object hierarchical named))
  (edges))
(defclass* edge (entity)
  ((edge-type :index :any) in-node out-node))
```

```
(defclass* attribute (q-object named)
  (elements entity expression touched))
(defclass* attribute-element (q-object)
  ((time-interval :initform (cons 0 0)) (value :initform "") attribute))
(defclass* document-quote (db-object)
  (document qobject start-pos end-pos))
(defclass* abstract-document (db-object named hierarchical)
  ())
(defclass* folder (abstract-document)
  ())
(defclass* document (abstract-document)
  ((content :initform "") quotes))
```

A.2 CLOS Constructors and Destructors

```
(defmethod initialize-instance :after
           ((self edge) &key
             name in-node out-node (spidr *spidr*)
            (type-parent (and (root-edge spidr)
            (edge-type (root-edge spidr)))))
  (when name
    (if type-parent
        (let ((et (retrieve-from-index 'edge-type 'name name)))
          (setf (edge-type self)
        (or et (make-instance 'edge-type
        :name name :parent type-parent :spidr spidr))))
        (progn
          (setf (edge-type self)
     (make-instance 'edge-type :name name :spidr spidr))
          (setf (root-edge spidr) self)))
  (when (and in-node out-node)
    (push self (in-edges in-node))
    (push self (out-edges out-node))))
```

A.3 Model Evaluator

```
(defun set-expression (attribute expr)
  (setf (expression attribute) expr)
  (dolist (ae (elements attribute))
    (setf (value ae) nil)))

(defun fill-values (expr time)
  (cond ((stringp expr) (value (s (concatenate 'string expr "." time))))
        ((listp expr) (mapcar (lambda(e) (fill-values e time)) expr))
        (t expr)))

(defmethod evaluate ((attr attribute) &key (force nil))
  (let ((expr (expression attr)))
    (when expr
      (loop for e in (elements attr) do
            (when (or force (not (value e)))
              (setf (value e)
                    (eval (fill-values expr (time e)))))))))
```

A.4 Scanner and Parser

```
(defun spidr-lexer (cmd)
  (let ((tokens '((id "^[^\.>:=]+") (pdelim "^[\.]") (edelim "^>")
                  (adelim "^:") (equals "^=")))
        (text cmd) (pos 0))
    (lambda ()
      (dolist (token tokens)
        (multiple-value-bind (r i)
          (match-re (second token) (subseq text pos) :return :index)
          (if r
              (let ((match (subseq text (+ pos (car i)) (+ pos (cdr i)))))
              (progn (setq pos (+ pos(cdr i)))
                     (return (values (car token) match)))))))))

(defun select (class pnode &optional parent)
     (let* ((child-name (first pnode))
            (test-child (find-if (lambda (x) (equal child-name (name x)))
                                 (children parent)))
              (child (if test-child
                         test-child
                         (if create
 (make-instance class :name child-name :parent parent)
                             nil))))
           (if (cdr pnode)
              (select class (cdr pnode) child)
             child))))


(defun (select-edge (out et in)
 (let ((test-edge (find-if (lambda (e) (eq et (edge-type e)))
             (directed-edges out in))))
     (if test-edge
         test-edge
         (if create (make-instance 'edge :edge-type et :in-node in :out-node out)
         nil)))))


(defun parse (text &key (create nil) (spidr *spidr*))
 (let ((cmd (parse-with-lexer (spidr-lexer text) *spidr-parser*)))
(case (first cmd)
   ('node (select 'node (second cmd) (root-node spidr)))
   ('edge (let ((out (select 'node (second cmd) (root-node spidr)))
                (in (select 'node (fourth cmd) (root-node spidr)))
    (edge-type (select 'edge-type (third cmd) (edge-type (root-edge spidr)))))
            (when (and out in)
              (select-edge out edge-type in))))))
```

Bibliography

[1] J. Bartolomei, R. Neufville, D. Hastings, and D. Rhodes. Screening for real options in an engineering system: A step towards flexible system development.

[2] Jason Bartolomei. *Qualitative Knowledge Construction for Engineering Systems: Extending the Design Structure Matrix Methodology in Scope and Procedure.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2007.

[3] Berkeley db.

[4] Peter Pin-Shan Chen. The entity-relationship model toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[5] W. M. Cohen and D. A. Levinthal. Absorptive capacity: A new perspective on learning and innovation. *Administrative Science Quarterly*, 35:128–152, 1990.

[6] M. Danilovic and T. R. Browning. Managing complex product development projects with design structure matrices and domain mapping matrices. *International Hournal of Management*, 25:300–314.

[7] Franz inc.

[8] Heidi Gregersen and Christian S. Jensen. Temporal entity-relationship models—a survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(3), 1999.

[9] H. B. Koo. *A Meta-language for Systems Architecting.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2005.

[10] OMG. Unified modeling language (uml), 2003.

[11] The jython project.