A Constraint Framework for Analysis of Engineering Systems

by

Shannon Jun Ho Iyo
S.B., C.S. M.I.T., 2006

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

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Feb 2008

Author .... [illegible] ....
Department of Electrical Engineering and Computer Science
January 25, 2008

Certified by .... [illegible] ....
Daniel E. Hastings
Professor of Engineering Systems and Aeronautics and Astronautics
Thesis Supervisor

Accepted by .... [illegible] ....
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Students

A Constraint Framework for Analysis of Engineering Systems

by

Shannon Jun Ho Iyo

Submitted to the Department of Electrical Engineering and Computer Science
on January 25, 2008, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

This thesis presents the design and implementation of a constraint framework for use in analyzing engineering systems. This framework extends the Engineering Systems Matrix to allow the definition of quantitative relationships between model objects. The addition of formal quantitative constraints allows analysts to make use of new techniques and simplifies maintenance of the data model.

Thesis Supervisor: Daniel E. Hastings
Title: Professor of Engineering Systems and Aeronautics and Astronautics

Acknowledgments

I would like to thank Professor Hastings for his patience and support of this project. Also thank you to Jason Bartolomei and Igor Sylvester for their hard work and good ideas upon which this thesis is built. And finally thank you to Reesa Phillips, Jennifer Wilds, David Broniatowski, and Nirav Shah for all the hard work, advice, and help.

Contents

1 Introduction **11**

1.1 Background . . . 12

2 Engineering Systems and the ESM **13**

2.1 Engineering Systems . . . 13

2.2 The Engineering Systems Matrix . . . 14

3 The Frog Analysis Tool **17**

3.1 System Design . . . 18

3.1.1 Data Representation . . . 18

3.1.2 Naming Conventions . . . 18

3.2 System Implementation . . . 19

3.2.1 Server Architecture . . . 22

4 The Constraint Framework **27**

4.1 Design . . . 27

4.1.1 Equation Syntax . . . 28

4.2 Implementation . . . 30

4.2.1 Data Representation . . . 30

4.2.2 Additional Server Commands . . . 32

4.2.3 Equation Evaluator . . . 33

4.2.4 Constraint Invariants . . . 34

4.2.5 Constraint Graphical User Interface . . . 35

5 Conclusion **43**

5.1 Contributions 43

5.2 Known Issues 43

5.3 Future Work 44

A Selected server source code **47**

A.1 Spidr.java 47

A.2 SpidrEnvironment.java 58

A.3 SpidrDatabase.java 61

A.4 Attribute.java 62

A.5 Constraint.java 67

B Selected user interface source code **73**

B.1 ConstraintPanel.java 73

B.2 ConstraintListCellRenderer.java 75

B.3 CreateConstraintDialog.java 76

B.4 EquationTextPane.java 81

List of Figures

2-1 The Engineering Systems Matrix 15

3-1 Inheritance property of nodes 19

3-2 Example of naming conventions 20

3-3 Frog application architecture 21

3-4 Class structure of the Frog server data model 23

4-1 The constraint list user interface 37

4-2 The first constraint input interface design 38

4-3 The drop-down list in the first constraint input interface 38

4-4 The second constraint input interface design 39

4-5 Auto-complete step 1 40

4-6 Auto-complete step 3 40

4-7 Auto-complete step 3 40

5-1 The search box 45

5-2 The ESM view 46

Chapter 1

Introduction

Systems in the real world are often large and complex, with many interacting components, and because of this there are many interesting real-world systems that are not understood in great detail. For example, a restaurant is far from the most complicated of but is quite complex nonetheless. There are a number of people involved, all performing different tasks and interacting with each other and with people outside the organization to reach a common goal. In order to make better decisions in situations involving complex real-world systems, it is important to try and understand such systems in greater depth. For example, one might study the restaurant in order to determine which people or positions are the critical links in the restaurant's day-to-day processes, and reassign duties appropriately. This thesis describes the extension of a methodology for performing this kind of analysis.

Chapter two of the thesis describes the Engineering Systems Matrix (ESM) methodology for capturing a general system, which is used to model real-world systems for formal analysis.

Chapter three covers the design and implementation of the Frog software tool, which is a Java-based application used to analyze ESMs.

Chapter four is about the design and implementation of the constraint framework, which adds numerical capabilities to the ESM and to the Frog application. It also describes the user interface that was created for interacting with the constraint framework.

Chapter five discusses the contributions made by this thesis, lists known issues, and describes possible future directions of research.

1.1 Background

The work executed in this thesis is built upon previous work at the Massachusetts Institute of Technology. Bartolomei's research on the formalization and analysis of complex engineering systems [1] via the Engineering Systems Matrix (ESM) lays the foundation for this area of research. Sylvester's work on the hierarchical representation of knowledge [7] provides a computer framework and implementation of Bartolomei's ESM methodology. This thesis project extends and improves Sylvester's implementation of the Frog analysis tool for complex engineering systems.

Frog is a system analysis application designed to address the difficulties engineering system analysts regularly face when creating, organizing, managing, and interpreting large collections of data acquired during field research. From the application viewpoint, these tasks translate into entering, editing, storing, and analyzing data. The Frog application immensely expedites the entry and editing of engineering system data, and it is a simple task for a computer program to maintain the data once entered. However, Frog is lacking in tools for analysis once these tasks are completed. It only provides the most basic analysis assistance in the form of the basic tree and table views. When analysis is to be performed, the data is often exported to be analyzed using Excel or Matlab.

The goal of the work presented here is to provide analytical functionality in the ESM and the Frog application. To accomplish this, I present the design and implementation of a numerical capability for Frog. This is achieved through the addition of a constraint framework that allows user-defined constraints between different components in system model. The constraint framework adds a formal, quantitative layer to the Frog software tool, making the model more flexible and enabling new analysis possibilities.

Chapter 2

Engineering Systems and the ESM

2.1 Engineering Systems

The artificial world is increasing in interconnectedness and complexity. Accordingly the understanding, analysis, and management of real-world systems is also increasing in difficulty. In the past, there has been research involving various types of complex systems in economics, ecology, politics, and other many fields [2][6]. However, until recently complex systems involving interactions between domains have remained unstudied. This is where the concept of the engineering system comes into play.

In general, an engineering system is a formalized representation of a real-world system. It fits the real-world system to a defined framework and in doing so enables the analysis of real-world systems. Engineering systems can be described as networks of social, technical, functional, and process components that combine and interact to create an outcome.

As a concrete example, an engineering system could be used to represent a restaurant. The social components would represent the people managing, working, and dining at the restaurant. The technical components would model cash registers, computer systems, appliances and other physical parts of the system. The functional components would model tasks or goals of the restaurant or parts of the restaurant, such as maintaining a sanitary kitchen and dining room conditions or providing good service to diners. The process components would describe activities performed to achieve the stated functions, such as a kitchen helper washing dishes and mopping the floor to preserve sanitation or a server checking on diners to ensure customer satisfaction.

Another real-world system that an engineering system could describe is a software company designing a new computer application. This engineering system would consist of engineers, managers, company executives, shareholders, and computer systems, as well as all of the functions and processes associated with such a task.

2.2 The Engineering Systems Matrix

As the above examples illustrate, an engineering system is able to model a wide range of real-world systems without being tethered by rigidly defined modeling rules and this makes it a powerful concept. However this also leaves open the question of exactly how to model the general engineering system. In his PhD thesis work, Bartolomei proposes a novel solution to this problem with the Engineering Systems Matrix (ESM), which is a basis for this thesis project.

The ESM is a framework for modeling and analyzing an engineering system. It formalizes the engineering system as a directed graph of nodes and edges. The components are represented by nodes in the graph. Since there are different types of components represented in an engineering system (social, technical, functional or process), each node has a category. This is beneficial for analysis as it organizes the model into groups of similar components that have an intuitive meaning to the analyst. Interactions and relationships between components are represented in the ESM by labeled directed edges. The label on an edge describes the relationship it represents. For example, in the engineering system modeling a software company, edges could model chain-of-command from executives down to engineers (Person X supervises Person Y), or an individual's duties (Person X responsible for Task Y).

In real-world systems, components and interactions may have a near infinite range of properties. In the ESM methodology these properties are captured as descriptive annotations called attributes. Attributes can be thought of as variables that define system components or interactions. Each node and edge in the graph may have any number of attributes. Attributes consist of a name and value(s) that are either qualitative or quantitative. A software engineer might be modeled with attributes of salary or programming languages known. A flow relation might have an attribute describing its rate.

In addition, the ESM is also able to capture the temporal aspects of real-world systems. Since real systems can and almost invariably will change over time, each attribute value is annotated with a time period. Such a pairing of an attribute value and time period is referred to as an attribute element. Each attribute may have multiple, non-temporally-overlapping attribute elements corresponding to different values during different time periods. This allows system analysis to be sensitive to time-dependent dynamics. For example, a software engineer may have a salary attribute, which can be expected to have multiple values resulting from raises and promotion.

Figure 2-1 below shows the ESM class diagram.

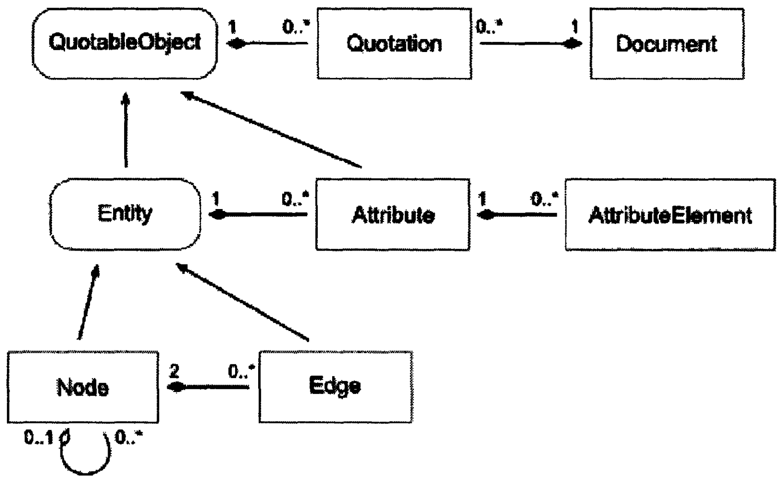


Figure 2-1: The Engineering Systems Matrix models an engineering system using graph objects.

Chapter 3

The Frog Analysis Tool

For an analyst modeling a real-world system with an engineering system, the major tasks fall into three categories:

1. Collection of data

2. Organization of data into system form

3. Analysis of data

Since engineering systems are a relatively new area of research, there are few if any existing tools to assist in these tasks. Bartolomei and Sylvester recently collaborated to create a software tool, known as "Frog," designed specifically to facilitate the second and third items on the list: construction and analysis of engineering systems. Frog is based on the ESM model described earlier, and enables engineering system analysts to easily create, organize, manipulate, and view an ESM model. The Frog program is the basis on which the constraint framework described in this thesis is implemented.

3.1 System Design

3.1.1 Data Representation

Frog uses a specific data representation of the ESM. The application's ESM data model is designed in a hierarchical manner, as described by Sylvester in his Master's of Engineering thesis. Nodes in the ESM are represented as nodes in a rooted acyclic graph, also known as a tree. The key advantage to representing nodes in this hierarchical manner is that it allows inheritance from ancestral nodes. In this hierarchical framework all attributes of a node are inherited by the node's children, and are inherited in turn by the children of those nodes. This allows for the grouping of related components and for the abstraction of common properties in such components.

There is a single root node in every ESM, from which all other nodes are descendent. The root node has an attribute called "existence" that represents the node's existence. The root node itself is not a tangible component in the engineering system so its existence is undefined in that context. However, because of the inheritance property of the representation, all subsequent nodes inherit the existence attribute. Thus each node in the system has a defined period or periods of existence (according to the time-dependent properties of attributes as described earlier). Figure 3-1 shows the inheritance property of nodes.

3.1.2 Naming Conventions

In his research, Sylvester defined conventions for referring to data objects in the system. These conventions are described here.

- Nodes are specified according to their path from the root node, with the names of nodes along the path separated by the '.' character:

  [node path] = [parent path].[node name]

  e.g. node.child.grandchild

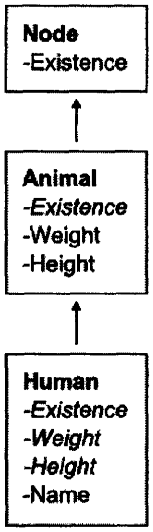


Figure 3-1: An example of the inheritance property of nodes. The root node, "Node," has the attribute "Existence," which is passed down to its child "Animal." "Animal" has two more attributes, "Weight" and "Height." These two attributes, as well as the inherited "Existence" attribute, are passed down to its child, "Human."

- Relations are stated as the name of the source node followed by the '>' character, followed by the relation name, another '>' character, and finally the target node name:

  [relation path] = [source node path]>[relation name]>[target node path]
  e.g. node.child1>relation>node.child2

- Attributes are stated as the path of the node or relation that the attribute describes, followed by a ':' character and then the name of the attribute:

  [relation path] = [owning node or relation path]:[attribute name]
  e.g. node.child:attribute

Figure 3-2 below shows an example of naming conventions.

3.2 System Implementation

The Frog application is designed according to a client-server architecture. A single backend application serves data via the Extensible Markup - Remote Procedure Call

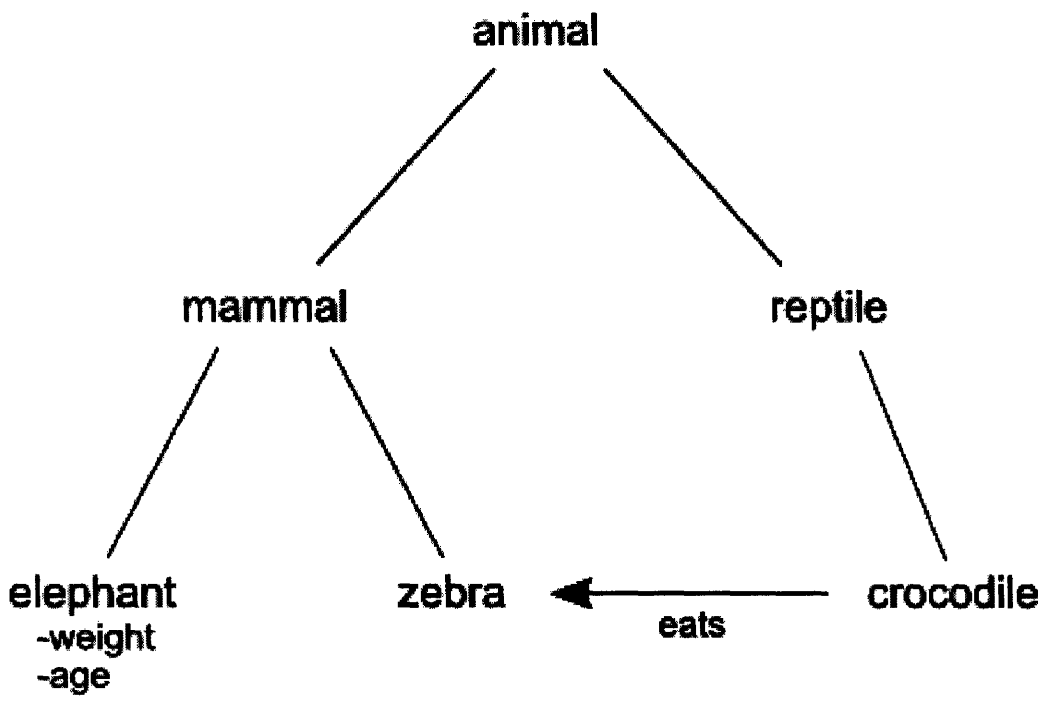


Figure 3-2: An example object model illustrating naming conventions. There are a total of 6 nodes: "animal", "mammal", "reptile", "elephant", "zebra", and "crocodile". There is an "eats" relation between "crocodile" and "zebra", and "elephant" has two attributes. The "elephant" node's "age" attribute is described as *animal.reptile.crocodile:age*. The "eats" relation that exists between the "crocodile" and "zebra" nodes would be described as *animal.reptile.crocodile>eats>animal.mammal.zebra*.

(XML-RPC) protocol to one or more remote frontend applications as shown in Figure 3-3 below. This facilitates centralized data storage and allows flexible remote access from any PC. In addition it allows multiple users to work on the same data.

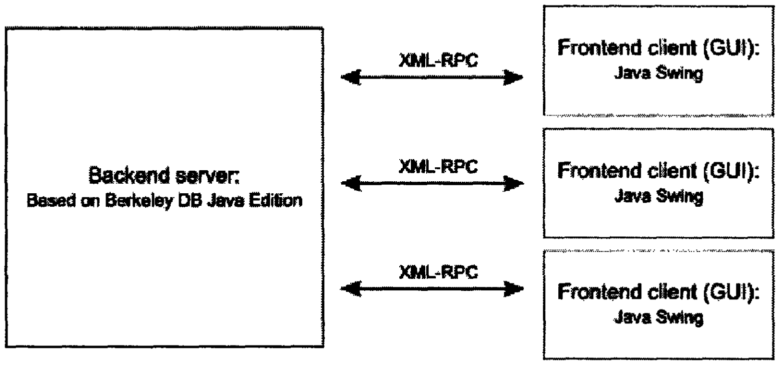


Figure 3-3: The Frog application has a client-server architecture. Requests are made to the server via an XML-RPC protocol.

The Frog client application is written in Java using the Swing GUI toolkit to provide platform independence as well as quick prototyping and iteration. The original server application was implemented in Common Lisp using the AllegroGraph semantic web library by Franz, Inc [5]. However, upon further consideration, it was realized that a server implemented using AllegroGraph presented some problems:

- AllegoGraph is proprietary software and thus each machine running a Frog server would require a license from Franz.
- Each developer's machine would require a license from Franz in order to compile the Frog server.
- The client and server applications being written in two different programming languages (Java and Common Lisp, respectively) might hinder future development efforts.

Since this thesis project involved additions to the server, it was decided to reimplement the server from the ground up based on a persistent database library that would avoid the problems AllegroGraph caused. The ideal library would be reliable, efficient on widely varying scales of operation, non-proprietary, and compatible with

Java. Towards these goals, Oracle's Berkeley DB Java Edition [3] was chosen. Berkeley DB is a mature open-source embeddable database library. It provides a persistent database that is reliable, and has a transaction and logging framework which allows backup and data recovery in the event of a problem. Berkeley DB is also efficient, having been proven to perform quite well on small and large datasets. Lastly, it is open-source so there are none of the licensing problems associated with AllegroGraph or other proprietary database libraries.

3.2.1 Server Architecture

The server consists of three main parts: the Spidr class, which handles ESM operations, the database wrapper for Berkeley DB, and the XML-RPC server implemented using the Apache XML-RPC library [4]. In addition for each type of object in the ESM there is a Java class for conversion between Java data and Berkeley DB records (see Appendix A for selected server source code).

The Spidr class is implemented as `frog.server.Spidr.java` and provides all read and write operations on the ESM data model. The Spidr class owns all model objects and manages their interaction with the environment and database.

The Berkeley DB wrapper is implemented in `frog.server.SpidrEnvironment.java` and `frog.server.SpidrDatabase.java`. These classes provide methods for setting up, reading and writing records on Berkeley DB databases. Common data members and methods are stored in the abstract classes `StateObject`, `DatabaseObject`, `QuotableObject`, and `Entity`. All other data classes inherit from these abstract classes, as shown in Figure 3-4 below:

The XML-RPC server is written as `frog.server.RPCServer.java` and handles remote client requests, forwarding calls onto the Spidr. It provides the following remote procedure calls:

- `getSpidrs()`: Returns the UIDs of all ESMs.

- `makeSpidr(String name, String[] classNames)`: Creates a new ESM with the

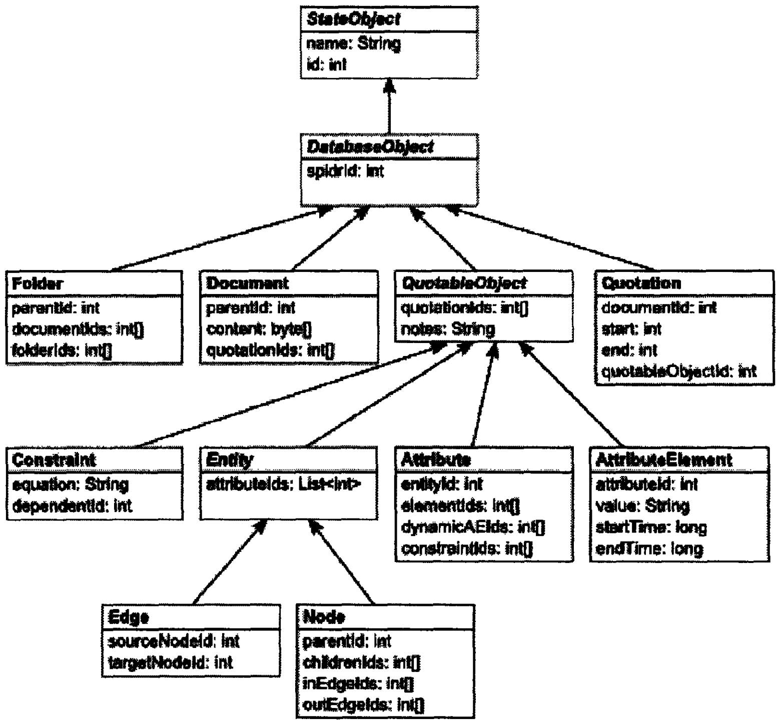


Figure 3-4: Class structure of the Frog server data model. The abstract (non-instantiable) classes are shown in italic font. All model classes inherit data members from their parents and ancestors. For example, the Edge class inherits from its ancestors StateObject, DatabaseObject, QuoatableObject, and Entity. Thus it has the member fields name, id, spidrId, quotationIds, notes, attributeIds, sourceNodeId, and targetNodeId.

given class names. Returns the UID of the new ESM.

- `removeSpidr(int esmId)`: Deletes the specified ESM. Returns true if the removal succeeded.

- `get(int id)`: Returns a String-keyed map of the relevant fields for the ESM object specified by `id`. If there is no such object, returns null.

- `remove(int esmId, int id)`: Removes the specified object from the ESM specified by `esmId`. ESM objects owned by the removed object are also removed, and ESM objects owning the removed object are updated appropriately. Returns true if the removal succeeded.

- `makeNode(int esmId, String name, int parentNodeId)`: Creates a new child of the node specified by `parentNodeId`. Returns the new node's UID, or *NULL_ID* [1] if the node could not be created.

- `setPositionOf(int esmId, int parentNodeId, int childNodeId, int position)`: Sets the child order of a node.

- `makeEdge(int esmId, String name, int sourceNodeId, int targetNodeId)`: Creates a new relation between the nodes specified by `sourceNodeId` and `targetNodeId`. Returns the new relation's UID, or *NULL_ID* [1] if the relation couldn't be created.

- `makeAttribute(int esmId, String name, int entityId)`: Creates a new attribute for the node or relation specified by `entityId`. Returns the new attribute's UID, or *NULL_ID* [1] if the attribute couldn't be created.

- `makeAttributeElement(int esmId, int attributeId)`: Creates a new element for the attribute specified by `attributeId`. The new attribute element will not be initialized with start and end times or a value. Returns the new element's UID, or *NULL_ID* [1] if the element couldn't be created.

[1]The UID *NULL_ID* is a special reserved constant that represents the ID of a non-existent data object.

- `setTimeInterval(int esmId, int elementId, long startTime, long endTime)`: Sets start and end times for the element specified by `elementId`. Returns true if the operation succeeded.

- `setValue(int esmId, int elementId, String value)`: Sets the value for the element specified by `elementId`. Returns true if the operation succeeded.

- `makeDocument(int esmId, String name, int folderId)`: Creates a new document in the folder specified by `folderId`. Returns the new document's UID, or *NULL_ID* [1] if the document couldn't be created.

- `setContents(int esmId, int docId, byte[] value)`: Sets the contents for the document specified by `docId`. Returns true if the operation succeeded.

- `getContents(int esmId, int docId)`: Returns the contents of the document specified by `docId`.

- `makeFolder(int esmId, String name, int parentId)`: Creates a new folder in the parent folder specified by `parentId`. Returns the new folder's UID, or *NULL_ID* [1] if the folder couldn't be created.

- `makeQuotation(int esmId, int qObjId, int docId, int start, int end)`: Creates a new quotation for the node, relation, attribute, or attribute element specified by `qObjId` in the document specified by `docId`. The parameters `start` and `end` indicate what part of the document the quotation references. Returns the new quotation's UID, or *NULL_ID* [1] if the quotation couldn't be created.

- `parse(int esmId, String query, boolean createIfNotFound)`: Searches for an object in the given ESM exactly matching `query`. The parameter `createIfNotFound` specifies if the object should be created in the case that it is not already present. Returns the UID of the object, or *NULL_ID* [1] if no matching object was found.

- `query(int esmId, String query)`: Searches for objects in the given ESM that partially match `query`. Returns an array of UIDs of objects that have names containing the query. The returned array will be empty if no matching objects were found.

Chapter 4

The Constraint Framework

The Frog analysis tool greatly assists analysts in managing and working with the datasets of complex engineering systems. However it was previously lacking in quantitative analysis abilities. Researchers desiring to perform numerical analysis on an ESM created in Frog would need to export data to Matlab, Excel, or other mathematics-enabled programs. In these cases the Frog application was relegated to an elaborate data store. The constraint framework tackles the problem by enabling exactly the kind of formal quantitative analysis that was previously absent from Frog. By adding mathematical relationships to ESM objects, it facilitates the modeling of natural constraints between distinct properties in an engineering system, and gives Frog a quantitative analysis ability. With the framework for constraints in place, it is possible to change values in one part of the ESM and observe the resulting changes throughout the model.

4.1 Design

The idea behind the constraint framework is to allow mathematical equations between attributes in the model. In this framework, a constraint is defined as a mathematical relationship between the values of two or more attributes. Attributes in a constraint may be *dependent* or *independent*. A valid constraint must have exactly one dependent attribute and one or more independent attributes, as well as a valid equation relating these attributes.

A constraint has two important values: its equation and the dependent attribute. The equation specifies which attributes are affected by the constraint, as well as the mathematical relationship between them. The equation is defined when the constraint is created and may not be changed. This decision highlights the equation as the definitive property of the constraint; if the equation were to be altered the constraint would no longer be the same constraint. Thus the design of the framework requires that if the equation relating a set of attributes must be changed, the constraint relating them should be deleted and a new constraint created. In contrast, the single dependent attribute in the equation may be changed at any time to one of the other attributes in the equation.

Implementations of the constraint framework must be dynamic, reflecting changes in values of independent attributes by updating the values of the dependent attributes accordingly. In addition, implementations must also protect against conflicting constraint states, which would result in attribute values being undefined. Such a state could occur if the same attribute was a dependent variable in two conflicting equations, causing the attribute to have two different possible values at any given time. Another situation where a conflicting state would occur is if there was a dependency loop in the constraint framework (two or more dependent attributes that depend either directly or indirectly on each other), which would make it impossible to calculate any of the dependent values.

4.1.1 Equation Syntax

To make the constraint framework a computational framework, a well-defined syntax for defining constraint equations is needed. In order to simplify the parsing and evaluation of equations, equations are specified the same way equations are written in Lisp syntax. There are two main differences between this equation syntax and typical mathematical syntax. The first and most notable difference is that the mathematical operator must always occur at the beginning of a statement, prior to all operands. The second is that every statement must be enclosed in parentheses — '(' and ')'.

The parentheses also specify the grouping and order of operations.

Here is a formal definition of a constraint equation:

> equation := ('=' *statement*$_1$ *statement*$_2$),
>
> where
>
> statement := (operator *operand*$_1$ *operand*$_2$ ... *operand*$_n$),
>
> operator ∈ {'+', '-', '*', '/', ...},
>
> operand ∈ *statement* ∪ *variable* ∪ $\Re$, and
>
> variable := '?'*variableName*

The current implementation in general is not able to evaluate equations with repeat instances of the same variable. In addition, it only supports two-operand statements and a limited set of operators, so the working syntax is limited to:

> **equation := ('=' *statement*$_1$ *statement*$_2$),**
>
> **where**
>
> **statement := (operator *operand*$_1$ *operand*$_2$),**
>
> **operator ∈ {'+', '-', '*', '/'},**
>
> **operand ∈ *statement* ∪ *variable* ∪ $\Re$, and**
>
> **variable := '?'*variableName***

For example, the following equation written in normal math syntax:

x = *3y + 1*

would be written in the Frog syntax as:

(= ?x (+ (3 ?y) 1))*

The operands are allowed to occur in any order, thus the following two statements are equivalent:

(= ?x (+ (3 ?y) 1))*

(= (+ 1 (?y 3)) ?x)*

Constraint equations are defined according to this syntax, with the unique IDs of attributes serving as variables. So a constraint specifying equality between the attributes with IDs 1041 and 1042 would be represented in the backend with the following equation:

(= ?1041 ?1042)

4.2 Implementation

The constraint framework was implemented in Java as an extension of the Frog analysis application. The implementation of the constraint framework required changes to the server application as well as the client application. In the server additional model classes were added and modified, and an evaluator for constraint equations was written. In the client corresponding model changes were implemented, and a new constraint graphical user interface was added.

4.2.1 Data Representation

In order to implement the constraint framework, some changes were made to the data representation in the server. A new data class representing a constraint was added, and the attribute data class was modified to allow dynamically computed attribute elements. The constraint class stores the vital data of its equation as well as the ID of the dependent attribute. It is also possible to find the IDs of the independent attributes by calling `frog.server.model.Evaluate.getVariableIds` on the equation and then removing the dependent attribute's ID from the returned list.

Attributes were augmented to store two types of attribute elements: static and dynamic elements. Static elements are the standard user-defined attribute elements, and are always present. Dynamic elements are the elements that result from a constraint, and are present only if it is a dependent attribute. Keeping two different lists of attribute elements preserves the static elements, which is important because attributes can switch between being dependent or independent. This happens when a constraint is created or removed, or when the dependent attribute of a constraint is switched. When the server receives a `get` request for an attribute, the attribute element data that is returned uses the dynamic elements if they exist, and the static elements if there are no dynamic elements.

Each attribute also stores a list of all constraints that involve it. This is important because dependent attributes must be updated if one of the attributes they depend upon changes. When an attribute's elements are added, removed, or modified, the server recomputes the elements of the dependent attribute for each constraint involving the modified attribute. This change may propagate through dependent attributes that appear as independent variables in other constraints, so the update is recursively propagated as needed through all affected attributes.

Because attributes are time-dependent, finding the dynamic elements of an attribute is in general a non-trivial task. For a given constraint, the value of a dependent attribute is only computable on the time intervals when the independent variables all have defined values. Thus the dependent attribute will have elements for all time intervals that are the intersection of the time intervals for the independent attribute's elements. The dynamic elements are determined by finding all intersecting time intervals, and then for each interval evaluating the constraint equation using the attributes' values during that interval. Each evaluation produces a new value that is paired with the time interval to create a dynamic attribute element.

Take the following example of a constraint involving one dependent attribute and two independent attributes:

(= ?1001 (+ ?1002 ?1003))
Dependent: ?1001

The constraint specifies that the value of the dependent attribute is equal to the sum of the values of the two independent attributes. Now suppose that the first independent attribute element has two elements with values 100 and 200, and that the second attribute element has two elements with values 25 and 50, as shown below:

?1002
- 01/01/07-02/01/07: 100
- 03/01/07-06/15/07: 200
?1003

- 01/15/07-04/15/07: 25

- 05/15/07-12/01/07: 50

Upon examination, it is apparent that the dependent attribute will have three attribute elements corresponding to the intersecting time periods of the independent attributes' elements, and this is how the dynamic attribute elements should be computed:

?1001

- 01/15/07-02/01/07: 125

- 03/01/07-04/15/07: 225

- 05/15/07-06/15/07: 250

4.2.2 Additional Server Commands

Additional database operations were also add to the Frog server application in order to provide an interface for manipulating constraints:

- `makeConstraint(int esmId, String name, String equation, int dependentId)`: Creates a new constraint and sets the ID of the dependent attribute. Returns its UID.
- `setDependentAttribute(int esmId, int constraintId, int dependentId)`: Sets which of a constraint's attributes is dependent. Returns true if the operation succeeds.

Calling `get(int constraintId)` returns a String-keyed map of the constraint's data with the following notable keys (non-interesting keys are omitted):

- "equation": the equation defining the constraint
- "attributes": the IDs of all attributes related by the constraint, whether they are dependent or independent
- "dependent": the ID of the dependent attribute

4.2.3 Equation Evaluator

The equation evaluator is a static class implemented as `frog.server.model.Evaluate.java` (see Appendix A for source code). It is responsible for evaluating constraint equations. Equations are expected to adhere to the syntax rules described earlier. The evaluator provides the following methods:

- `getVariables(String expression)`: A utility method that parses an expression and returns a list of the variables found in the expression. Recall that variables are denoted by the '?' character (e.g. ?x).
- `getVariableIds(String expression)`: A utility method that parses an expression and returns a list of the IDs found in the expression. Assumes that all variables are numerical IDs (e.g. ?2043).
- `evaluate(String expression, Map<String, Double> bindings)`: Evaluates the given expression with the provided variable bindings. May update the bindings if the expression is an equation and contains an unbound variable (recall that an equation is an expression containing the '=' operator). The expression may contain no unbound variables if it is not an equation, and exactly one unbound variable if it is an equation.

The `evaluate` method is where most of the work occurs. When `evaluate` is called on an expression, the evaluator first tests if the expression is a number, and if so returns that value. It then checks if the expression is a variable, and if so returns the binding of that variable. If the expression is a statement, the operands are recursively evaluated and then the operator is applied to them, giving a value.

If the expression is an equation, the unbound variable is determined by checking the bindings. Then the equation is rewritten to solve for the unbound variable. This step is performed in the `solveEquality` helper method. When solving, it is sometimes necessary to reverse operations to isolate the unbound variable. Once the equation is solved for the unbound variable, `evaluate` is called on the other side of the equation and the value returned is used to create a new variable binding in the bindings table.

Here is an example of an evaluator call, given the following expression and bindings:

> *(= ?5001 (/ (+ ?5002 ?5003) 2))*
>
> *BINDINGS: {?5001=5, ?5003=3}*

The expression is an equation, so the evaluator checks the bindings and determines that ?5002 is the unknown variable. The equation is progressively solved for the variable ?5002:

> *(= (+ ?5002 ?5003) (* ?5001 2))*
>
> *BINDINGS: {?5001=5, ?5003=3}*
>
> *and then*
>
> *(= ?5002 (- (* ?5001 2) ?5003))*
>
> *BINDINGS: {?5001=5, ?5003=3}*

The solver stops when the unknown variable is isolated, and then the other side of the equation is evaluated with the given bindings:

> *(- (* ?5001 2) ?5003)*
>
> *BINDINGS: {?5001=5, ?5003=3}*

This returns a value of 7, and the evaluation process is completed by adding a binding to the bindings table. The method returns a value indicating success, and the caller can find the solved value of the unknown variable in the bindings table.

> *(= ?5002 7)*
>
> *BINDINGS: {?5001=5, ?5002=7, ?5003=3}*

4.2.4 Constraint Invariants

As mentioned earlier in the Design section, the implementation of the constraint framework must satisfy two key invariants in order to ensure data correctness:

- No attribute is the dependent attribute in more than one constraint (for example `x = y, x = z` violates this condition if `x` is the dependent attribute in both equations)

- There are no dependency loops in the constraint framework. In other words it is illegal for two attributes to either directly or indirectly depend on each other (for example `x = y + 1`, `y = x + 1` forms a dependency loop if `x` is the dependent attribute in the first equation and `y` is the dependent attribute in the second equation).

The first invariant is ensured at the time of constraint creation or modification. It is relatively simple to check for the first invariant because each attribute in the model stores a list of all constraints that involve it. When a new constraint is added or there is a command to change the dependent attribute, the new dependent attribute's list of constraints is checked to make sure it is not already a dependent attribute for another constraint, and if so, the command is aborted.

The second invariant is also ensured at the time of constraint creation or modification. In order to detect dependency loops, the server considers a graph of attributes connected by constraints. It then searches this graph from the new dependent attribute using a depth-first search with an extended list (to prevent re-traversing the same branches more than once). If the search encounters the new dependent attribute again, this indicates a loop and the command is aborted. If the search does not encounter the new dependent attribute again, the invariant is satisfied and the command may safely be executed.

4.2.5 Constraint Graphical User Interface

A challenge associated with the addition of a constraint framework was creating a user interface for creating, viewing, and editing constraints. Constraints add another level of complexity to the data model and application, and the user interface is critical in managing this added complexity without overwhelming or confusing the user. For simplicity and consistency, the frontend uses the same Lisp-based equation syntax that the backend does, but replaces UIDs with fully qualified attribute names, enclosed by square brackets ('[' and ']'). Thus in the user interface, the constraint syntax is formally defined as:

equation := ('=' $statement_1$ $statement_2$),

where

statement := ($operator$ $operand_1$ $operand_2$),

operator $\in$ {'+', '-', '*', '/'},

operand $\in$ *statement* $\cup$ *variable* $\cup$ $\Re$, and

variable := [*variableName*]

For example, if we defined an equality constraint between Node:attribute1 with ID 1041 and Node:attribute2 with ID 1042, the equation would appear in the backend and frontend in two different forms as shown below:

BACKEND: (= ?1041 ?1042)

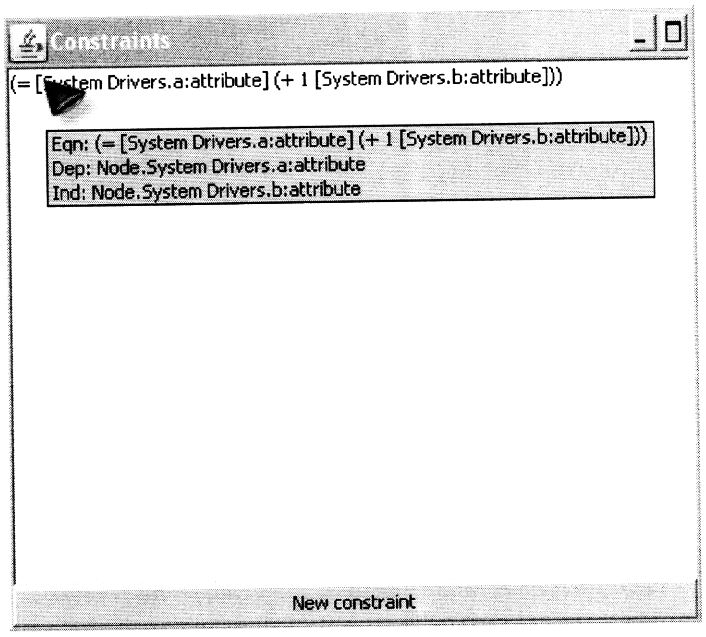
FRONTEND: (= [Node:attribute1] [Node:attribute2])

It is important to note that the evaluator in general will not work on equations with repeat instances of the same variable.

The conversion between the two representations is performed in the model class `frog.model.Constraint.java` (see Appendix A for source code). One side-effect of using this syntax is the addition of two new reserved characters. Since '[' and ']' have the special meaning of demarcating attribute names, they cannot be used in the names of model objects.

Currently the constraints for an ESM are displayed as a list of the defining equations (see Figure 4-1 below), with tooltip text appearing on mouse-over to show which attributes are involved and which is the dependent attribute. Double-clicking on a constraint in the list opens a dialog for viewing the constraint and setting the dependent attribute. Constraints may be deleted by selecting them and pressing the DELETE key. A dialog for creating a new constraint can be shown by pressing the "New constraint" button. This list will be redesigned to give a more information-rich view of the constraints present in the system.

The more complicated design problem was the question of how users would define and edit the equations in a constraint. The design process of the user interface for defining constraints has been dynamic, with two major design choices considered. In



(= [System Drivers.a:attribute] (+ 1 [System Drivers.b:attribute]))

Eqn: (= [System Drivers.a:attribute] (+ 1 [System Drivers.b:attribute]))
Dep: Node.System Drivers.a:attribute
Ind: Node.System Drivers.b:attribute

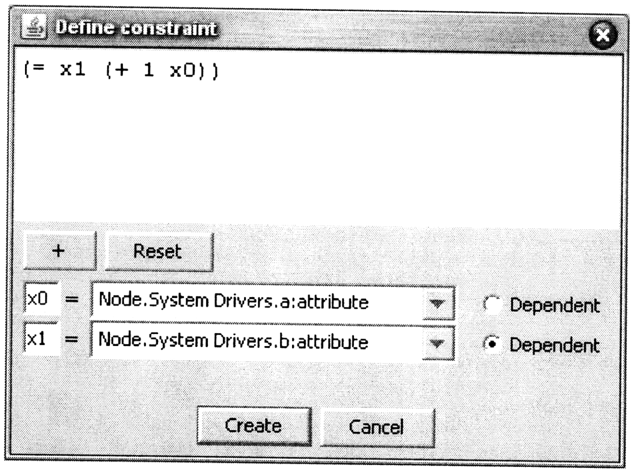
Figure 4-1: The constraint list user interface. The tooltip provides supplemental information including the names of the dependent and independent attributes.

both interfaces, the user enters an equation and specifies which attribute dependent. When the user presses the "Create" button, the fields are checked to ensure that they are valid, and if so, the constraint is created.

In the first design (shown in Figures 4-2 and 4-3 below) the problem is addressed by first assigning each attribute to a variable name (e.g. x, y, z) and then composing an equation from these variable names. The dependent attribute is selected by clicking one of the radio buttons to the right of the attribute names. Additional fields for variable bindings may be added by pressing the '+' button. A drop-down list in the attribute name entry fields assists the user in entering the attribute names by showing possible completions.

This design has the benefits of a succinct equation and assistance in entering attribute names. However it is not intuitive to use and it is easy for the user to enter an invalid equation by mistyping a variable name.

The second design (shown in Figure 4-4 below) which was ultimately chosen for use in the application simplifies the interface by having the fully qualified attribute names entered directly into the equation using square brackets to delimit attribute

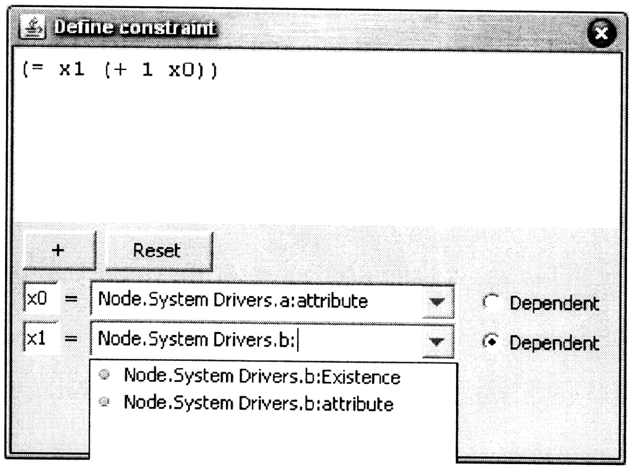


```
(= x1 (+ 1 x0))
```

x0 = Node.System Drivers.a:attribute [ ] Dependent

x1 = Node.System Drivers.b:attribute [✓ Dependent]

Figure 4-2: The first interface design. The variable x0 is assigned to the attribute Node.System Drivers.a:attribute and the variable x1 is assigned to Node.System Drivers.b:attribute



```
(= x1 (+ 1 x0))
```

x0 = Node.System Drivers.a:attribute [ ] Dependent

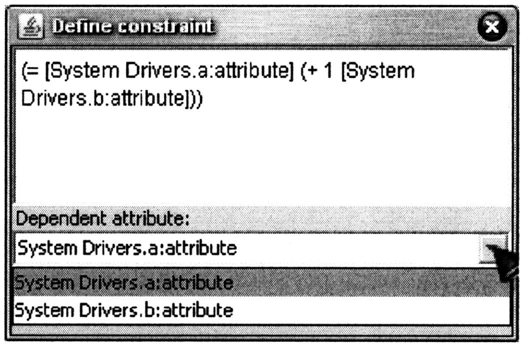
x1 = Node.System Drivers.b: [✓ Dependent]

Node.System Drivers.b:Existence

Node.System Drivers.b:attribute

Figure 4-3: The first interface design. A drop-down list box assists users in entering attribute names.

names from the rest of the mathematical symbols. The dependent attribute is selected via a dropdown list box populated according to the attribute names parsed from the equation.



```
(= [System Drivers.a:attribute] (+ 1 [System
Drivers.b:attribute]))
```

System Drivers.a:attribute

System Drivers.a:attribute

System Drivers.b:attribute

Figure 4-4: The second interface design. The user enters fully qualified attribute names directly into the equation, demarcated by square braces ('[' and ']'). The dependent attribute is selected from a drop-down list.

In order to simplify the entry of constraint equations, the equation area provides input assistance. There is an auto-complete feature that assists the user in entering attribute names by showing (in bold) the next possible completion for the name the user is typing. Pressing the TAB key will fill in the rest of the name with the auto-complete suggestion. Figures 4-5, 4-6, and 4-7 show the auto-complete function in action. In addition, when an open parenthesis or brace ('(' or '[') is typed, the matching closing parenthesis or brace (')' or ']', respectively) is automatically inserted so the user doesn't need to type it. These closing characters are also deleted if the matching open character is deleted.

The main weakness of the second interface design is that since all variable definitions are performed in a text entry area, there is no drop-down list of possible completions when entering an attribute name. This forces the user to recall the names of attributes that will be part of the constraint rather than just choosing them out of a list. Although the user receives assistance via the auto-complete feature, it is still desirable to show all (or at least several) possible completions instead of just one. It is likely that this issue can be addressed through additional development of a customized text area as is described in the Future Work section.

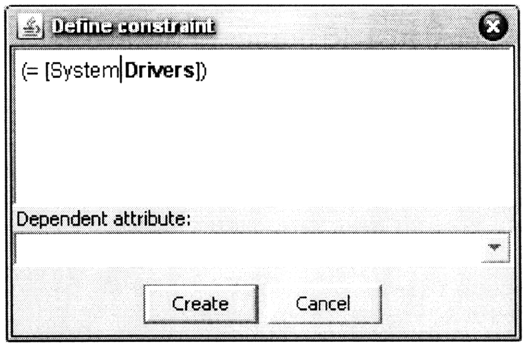


Figure 4-5: The second interface design showing auto-complete assistance. When the user begins typing a name, the next possible completion is shown in bold font.

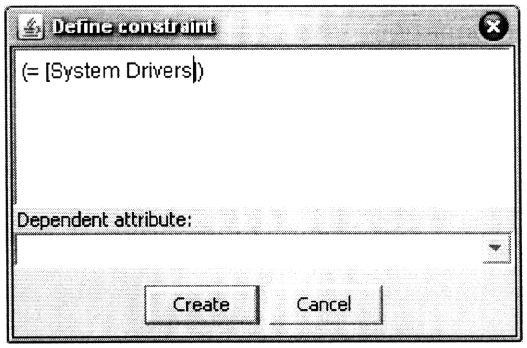


Figure 4-6: The second interface design showing auto-complete assistance. Pressing the TAB key fills in the rest of the name with the auto-complete suggestion.

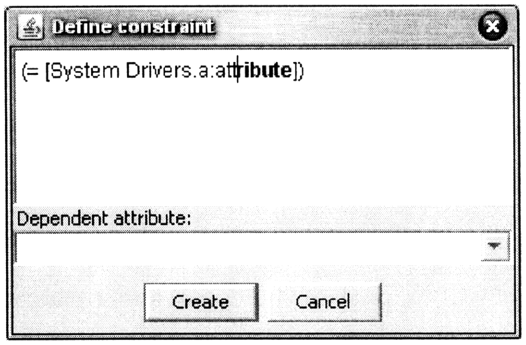


Figure 4-7: The second interface design showing auto-complete assistance. The user may then continue to enter the rest of the fully qualified attribute name with auto-complete assistance.

Even with the issue faced by the second interface, it was determined that its advantages of being more intuitive and less confusing outweighed the negatives, and so it was chosen over the first design. See Appendix B for the current source code of the GUI components described here.

Chapter 5

Conclusion

5.1 Contributions

The work presented here extends the Frog analysis tool and adds a mathematical capability to the Engineering Systems Matrix model. The constraint framework that was designed in this thesis gives new capabilities to the ESM, allowing equations to relate different attribute values in the ESM, adding a new dimension to engineering system analysis. The framework was implemented as an addition to the Frog analysis tool and provides a rich user interface for viewing, creating, editing, and deleting constraints from the system model. This implementation is written in Java and uses the Oracle Berkeley DB and Apache XML-RPC libraries. The extended Frog application allows analysts to perform more complex analysis while simplifying data management. In summary, the mathematical and relational capability this new constraint framework provides gives users more powerful analysis options.

5.2 Known Issues

The following is a list of known issues in the implementation of the constraint framework. These issues should be addressed during future development of the framework.

- The constraint-enhanced database model is not backwards-compatible because it introduces the new reserved symbols '[' and ']' (which join the already reserved symbols '<' and '>'). Because of this it is conceivable that older ESMs imported into the new application may have improperly named objects. This could be solved by pre-processing old databases and adding an escape character to reserved symbols (this has the drawback of introducing another reserved symbol). The constraint framework also requires new XML-RPC methods. This means that the new client will not work correctly with previous versions of the server (old client versions will work with the new server but will not be able to use constraints).

- The export function in the frontend is not set up for handling constraints, so constraints will not be persisted to exported ESMs. The export function will need to be updated to include constraints in its output files.

- The evaluator is not able to evaluate equations with repeated instances of the dependent attribute, for example (= (+ ?x ?x) 6) can not be evaluated. A more powerful evaluator would be desirable to allow analysts to enter a larger variety of equations.

5.3 Future Work

There are a number of logical extensions and improvements that can be built upon the work presented here, both in the user interface and in the framework itself.

The constraint framework provides flexibility and power to analysts, and the client application needs a full-featured user interface to take advantage of it. When creating a constraint, specifying equations can be a troublesome task if the user doesn't remember the full names of the attributes involved. It would reduce cognitive load on the user if the GUI provided a contextual drop-down list of possible completions (similar to the search box shown in Figure 5-1 below) to assist users when entering attribute names.

In order to assist the user in managing a highly-constrained system with many inter-related attributes, I propose the addition of new views to the GUI. A table view

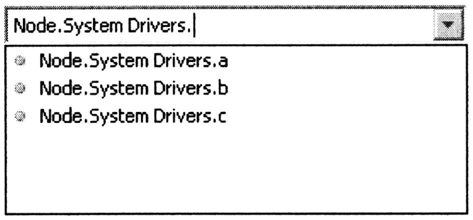
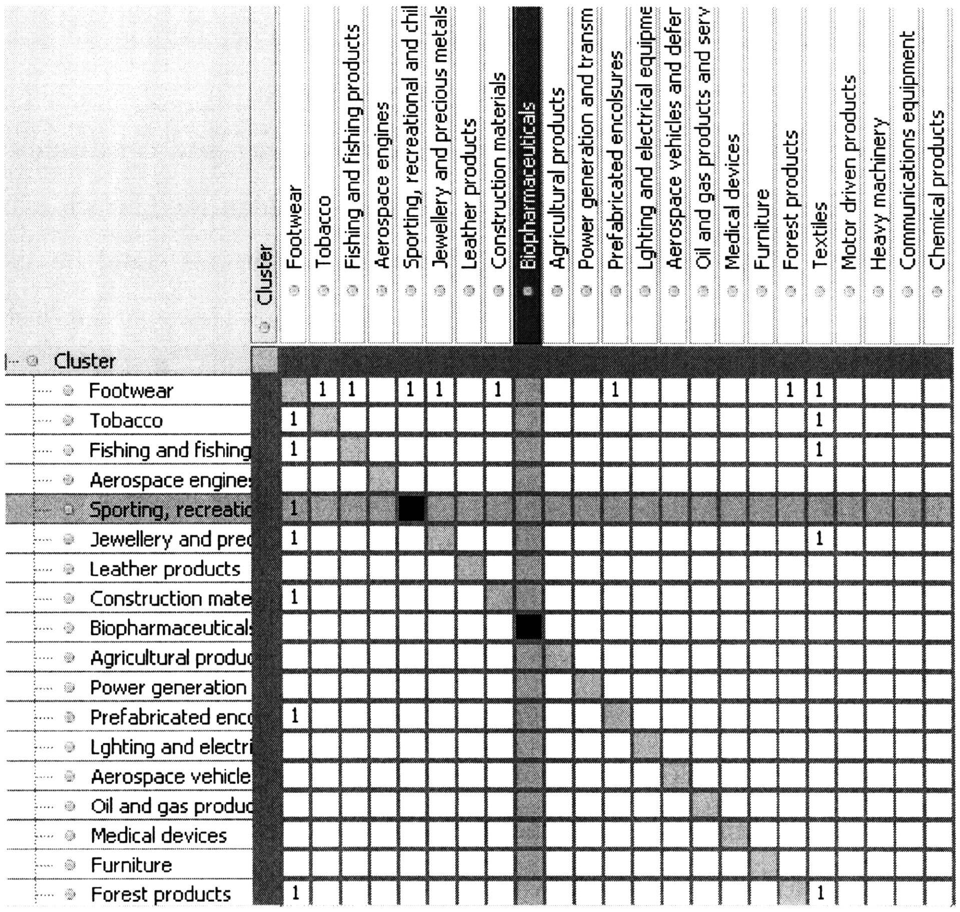


Figure 5-1: The search box provides a drop-down list of possible completions when entering names in the document coding editor.

similar to the ESM table view (shown in Figure 5-2 below) could give an immediate overview of how attributes in the model are co-constrained.

A graph view with attributes as the nodes in the graph and constraints as the edges in the graph would allow the analyst to see on a high level which attributes affect others the most. Dependent and independent attributes could be shown in different colors to distinguish between them. To add to this view, partial derivatives could be calculated to find how changes in an attribute's value affected neighboring attributes, and the chain rule applied to see the attribute's influence on further away attributes. These strengths of influence could be represented by thicker or thinner edges connecting attribute nodes. In this way it would be easy to determine which attributes in the graph are the most important in terms of influence.

In the backend the expression evaluator can be made more powerful and flexible. Adding more operator and variable types would allow a wider variety of equations. Boolean support would allow for interesting computation possibilities. A scripting language for plug-in functions is another possible way to expand the evaluator. This would allow functions to be defined by the user and included in constraint equations. To use such a scripting language the user would need to provide the definition of the function as well as its inverse so equations involving the scripted function could be solved in the evaluator.



	Cluster	Footwear	Tobacco	Fishing and fishing products	Aerospace engines	Sporting, recreational and chil	Jewellery and precious metals	Leather products	Construction materials	Biopharmaceuticals	Agricultural products	Power generation and transm	Prefabricated encolsures	Lghting and electrical equipme	Aerospace vehicles and defer	Oil and gas products and serv	Medical devices	Furniture	Forest products	Textiles	Motor driven products	Heavy machinery	Communications equipment	Chemical products
Cluster																								
Footwear			1	1		1	1		1				1						1	1				
Tobacco		1																		1				
Fishing and fishing		1																		1				
Aerospace engine																								
Sporting, recreatio		1																						
Jewellery and prec		1																		1				
Leather products																								
Construction mate		1																						
Biopharmaceutical																								
Agricultural produc																								
Power generation																								
Prefabricated enc		1																						
Lghting and electri																								
Aerospace vehicle																								
Oil and gas produc																								
Medical devices																								
Furniture																								
Forest products		1																		1				

Figure 5-2: The ESM view allows users to see which nodes are related to each other.

Appendix A

Selected server source code

A.1 Spidr.java

```
public class Spidr {

    // public static final int ROOT_NODE_ID = 1000;
    public static final int NULL_ID = -1;
    // IDs to return when there is a problem creating or editing constraints
    public static final int ATT_ALREADY_DEP_ERROR_ID = -2; // The attribute is
    // already dependent
    public static final int CYCLE_ERROR_ID = -3; // There is a dependency
    // cycle
    public static final int SPIDR_LIST_ID = 0;

    private Environment environment;
    // The main database
    private SpidrDatabase db;
    private IdFactory idFactory;

    private Transaction currentTxn = null;

    public Spidr(Environment env, Database objectDb, Database classDb,
            SecondaryDatabase namesDb, Database idDatabase) {
        this.environment = env;
        db = new SpidrDatabase(objectDb, classDb, namesDb);
        idFactory = new IdFactory(idDatabase);
        try {
            if (get(SPIDR_LIST_ID) == null) {
                createSpidrList();
            }
        } catch (DatabaseException e) {}
    }

    private int getNextId() {
        return idFactory.nextId(currentTxn);
    }

    public SpidrList getSpidrList() throws DatabaseException {
        return (SpidrList) get(SPIDR_LIST_ID);
    }
```

```
private Node createRootNode(int spidrId) throws DatabaseException {
    Node root = new Node(NULL_ID, "Node", getNextId(), spidrId);
    // Update the database
    db.newRecord(currentTxn, root.writeKey(), root.writeClass(), root
            .writeEntry());
    createAttribute(root.getId(), "Existence", true);

    return root;
}

private Folder createRootFolder(int spidrId) throws DatabaseException {
    Folder root = new Folder(NULL_ID, "Folder", getNextId(), spidrId);
    // Update the database
    db.newRecord(currentTxn, root.writeKey(), root.writeClass(), root
            .writeEntry());

    return root;
}

// Returns the DatabaseObject for the specified ID, or null if it isn't
// found
public StateObject get(int id) throws DatabaseException {
    if (id == NULL_ID) { return null; }
    DatabaseEntry key = StateObject.writeKey(id);
    return get(key);
}

// Returns the DatabaseObject for the specified key, or null if it isn't
// found
private StateObject get(DatabaseEntry key) throws DatabaseException {
    DatabaseEntry classEntry = new DatabaseEntry();
    DatabaseEntry objEntry = new DatabaseEntry();
    OperationStatus status = db.get(currentTxn, key, classEntry, objEntry);

    if (!status.equals(OperationStatus.SUCCESS)) return null;

    TupleInput in = new TupleInput(classEntry.getData());
    String className = in.readString();

    StateObject obj;
    if (className.equals(Attribute.class.getName())) {
        obj = new Attribute();
    } else if (className.equals(Constraint.class.getName())) {
        obj = new Constraint();
    } else if (className.equals(AttributeElement.class.getName())) {
        obj = new AttributeElement();
    } else if (className.equals(Document.class.getName())) {
        obj = new Document();
    } else if (className.equals(Edge.class.getName())) {
        obj = new Edge();
    } else if (className.equals(Folder.class.getName())) {
        obj = new Folder();
    } else if (className.equals(Node.class.getName())) {
        obj = new Node();
    } else if (className.equals(Quotation.class.getName())) {
        obj = new Quotation();
    } else if (className.equals(SpidrList.class.getName())) {
        obj = new SpidrList();
    } else if (className.equals(SpidrObject.class.getName())) {
        obj = new SpidrObject();
        } else {
            return null;
        }
        obj.readEntry(objEntry);
        return obj;
    }

    // Search for a node, relation, or attribute name in the database
    public List<DatabaseObject> query(int spidrId, String queryString)
            throws DatabaseException {

        List<DatabaseObject> objectList = new ArrayList<DatabaseObject>();
        SpidrObject spidrObj = (SpidrObject) get(spidrId);
        Node rootNode = (Node) get(spidrObj.getRootNodeId());
        // Check to make sure the name is prepended by the root node name
        if (queryString.indexOf(rootNode.getShortName()) != 0) {
            queryString = rootNode.getName() + Node.NODE_DELIMITER
                    + queryString;
        }

        // Check if it's an attribute
        String[] attributeSplit = queryString
                .split(Attribute.ATTRIBUTE_DELIMITER);
        if (attributeSplit.length == 1
                && queryString.contains(Attribute.ATTRIBUTE_DELIMITER)) {
            Entity entity = (Entity) parse(spidrId, attributeSplit[0], false);
            if (entity != null) {
                for (Integer id : entity.getAttributeIds()) {
                    objectList.add((Attribute) get(id));
                }
            }
        } else if (attributeSplit.length == 2) {
            Entity entity = (Entity) parse(spidrId, attributeSplit[0], false);
            if (entity != null) {
                for (Integer id : entity.getAttributeIds()) {
                    Attribute attribute = (Attribute) get(id);
                    if (attribute.getShortName().contains(attributeSplit[1])) {
                        objectList.add(attribute);
                    }
                }
            }
            return objectList;
        }

        // Check if it's an edge
        String[] edgeSplit = queryString.split(Edge.EDGE_DELIMITER);
        int edIndex = queryString.indexOf(Edge.EDGE_DELIMITER);
        if (edgeSplit.length == 3) {
            // source>edgeName>target: query on the node
            return query(spidrId, edgeSplit[2]);
        } else if (edIndex != -1) {
            // source>edgeName
            Node source = (Node) parse(spidrId, edgeSplit[0], false);
            if (source != null) {
                for (Integer id : source.getOutEdgeIds()) {
                    Edge edge = (Edge) get(id);
                    if (edge.getShortName().contains(
                            queryString.substring(edIndex + 1))) {
                        objectList.add(edge);
                    }
                }
            }
            return objectList;
        }

        // Check if it's a node
        int ndIndex = queryString.lastIndexOf(Node.NODE_DELIMITER);
        if (ndIndex != -1) {
            Node parent = (Node) parse(spidrId, queryString.substring(0,
                    ndIndex), false);
            if (parent != null) {
                for (Integer id : parent.getChildIds()) {
                    Node child = (Node) get(id);
                    if (child.getShortName().contains(
                            queryString.substring(ndIndex + 1))) {
                        objectList.add(child);
                    }
                }
            }
        }

        return objectList;
    }

    public DatabaseObject parse(int spidrId, String name,
            boolean createIfNotFound) throws DatabaseException {

        try {
            SpidrObject spidrObj = (SpidrObject) get(spidrId);
            Node rootNode = (Node) get(spidrObj.getRootNodeId());
            // Check to make sure the name is prepended by the root node name
            if (name.indexOf(rootNode.getShortName()) != 0) {
                name = rootNode.getName() + Node.NODE_DELIMITER + name;
            }

            DatabaseEntry nameKey = new DatabaseEntry(name.getBytes("UTF-8"));
            List<DatabaseEntry> candidateKeys = db.getInSecondary(currentTxn,
                    nameKey);

            for (DatabaseEntry keyEntry : candidateKeys) {
                StateObject so = get(keyEntry);
                if (so instanceof DatabaseObject) {
                    DatabaseObject dObj = (DatabaseObject) so;
                    if (dObj.getSpidrId() == spidrId) { return dObj; }
                }
            }

            if (createIfNotFound) {
                // Check if it's an attribute
                String[] attributeSplit = name
                        .split(Attribute.ATTRIBUTE_DELIMITER);
                if (attributeSplit.length == 2) {
                    Entity entity = (Entity) parse(spidrId, attributeSplit[0],
                            false);
                    if (entity != null) {
                        return createAttribute(entity.getId(),
                                attributeSplit[1]);
                    } else {
                        return null;
                    }
                }
            // Check if it's an edge
            String[] edgeSplit = name.split(Edge.EDGE_DELIMITER);
            if (edgeSplit.length == 3) {
                Node source = (Node) parse(spidrId, edgeSplit[0], false);
                Node target = (Node) parse(spidrId, edgeSplit[2], false);
                if ((source != null) && (target != null)) {
                    return createEdge(source.getId(), target.getId(),
                            edgeSplit[1]);
                } else {
                    return null;
                }
            }

            // Check if it's a node
            int ndIndex = name.lastIndexOf(Node.NODE_DELIMITER);
            if (ndIndex != -1) {
                Node parent = (Node) parse(spidrId, name.substring(0,
                        ndIndex), false);
                if (parent != null) {
                    return createNode(parent.getId(), name
                            .substring(ndIndex + 1));
                } else {
                    return null;
                }
            }

            // Can't parse it
            return null;
        } else {
            return null;
        }
    } catch (UnsupportedEncodingException willNeverOccur) {
        return null;
    }
}

public boolean setPositionOf(int spidrId, int parentId, int childId,
        int position) throws DatabaseException {
    Node parent = (Node) get(parentId);
    if (parent == null) {
        return false;
    } else if (parent.setPositionOf(childId, position)) {
        update(parent);
        return true;
    } else {
        return false;
    }
}

public boolean setDependentAttribute(int constraintId,
        int dependentAttributeId) throws DatabaseException {
    StateObject sObj = get(constraintId);
    if (!(sObj instanceof Constraint)) { return false; }
    Constraint con = (Constraint) sObj;

    Attribute dependent = (Attribute) get(dependentAttributeId);
    if (dependent == null) {
        return false;
    } else if (dependent.isDependentAttribute()) {
        // it is already a dependent attribute
        // System.out.println("Already dependent");
```

```
            return false;
        }

        Attribute oldDependent = (Attribute) get(con.getDependentAttributeId());
        oldDependent.clearDynamicAEIds();

        con.setDependentAttributeId(dependentAttributeId);
        if (Constraint.containsDependencyCycles(con, this)) {
            // System.out.println("Contains cycles");
            return false;
        }

        update(oldDependent);
        update(con);
        oldDependent.propagateUpdate(this);
        dependent.propagateUpdate(this);
        return true;
    }

    public boolean setTimeInterval(int attributeElementId, long start, long end)
            throws DatabaseException {
        StateObject sObj = get(attributeElementId);
        if (!(sObj instanceof AttributeElement)) { return false; }
        AttributeElement elt = (AttributeElement) sObj;

        elt.setPeriod(start, end);
        update(elt);
        ((Attribute) get(elt.getAttributeId())).propagateUpdate(this);
        return true;
    }

    public boolean setValue(int attributeElementId, String value)
            throws DatabaseException {
        StateObject sObj = get(attributeElementId);
        if (!(sObj instanceof AttributeElement)) { return false; }
        AttributeElement elt = (AttributeElement) sObj;

        elt.setValue(value);
        update(elt);
        ((Attribute) get(elt.getAttributeId())).propagateUpdate(this);
        return true;
    }

    // Create a new Attribute
    public Attribute createAttribute(int entityId, String name)
            throws DatabaseException {
        return createAttribute(entityId, name, false);
    }

    // Create a new Attribute
    public Attribute createAttribute(int entityId, String name,
            boolean creatingRootAttribute) throws DatabaseException {

        StateObject obj = get(entityId);
        if (!(obj instanceof Entity)) { return null; }
        Entity entity = (Entity) obj;
        String longName = entity.getName() + Attribute.ATTRIBUTE_DELIMITER
                + name;

        // Make sure it isn't already present in the spidr
        if (!creatingRootAttribute
            && parse(entity.getSpidrId(), longName, false) != null) { return null; }

    Attribute attribute = new Attribute(entityId, longName, getNextId(),
            entity.getSpidrId());
    entity.addAttributeId(attribute.getId());

    // Update the database
    db.newRecord(currentTxn, attribute.writeKey(), attribute.writeClass(),
            attribute.writeEntry());
    update(entity);

    // Inherit the attribute
    if (entity instanceof Node) {
        Node node = (Node) entity;
        for (Integer childId : node.getChildIds()) {
            createAttribute(childId, name);
        }
    }

    return attribute;
}

// Create a new AttributeElement
public AttributeElement createAttributeElement(int attributeId,
        String value, long startTime, long endTime, boolean isDynamic)
        throws DatabaseException {

    StateObject obj = get(attributeId);
    if (!(obj instanceof Attribute)) { return null; }
    Attribute attribute = (Attribute) obj;
    AttributeElement element = new AttributeElement(attributeId, value,
            startTime, endTime, getNextId(), attribute.getSpidrId());
    if (!isDynamic) {
        attribute.addAttributeElementId(element.getId());
        attribute.propagateUpdate(this);
    }

    // Update the database
    db.newRecord(currentTxn, element.writeKey(), element.writeClass(),
            element.writeEntry());
    update(attribute);

    return element;
}

public Constraint createConstraint(String name, String equation,
        int dependentAttributeId) throws DatabaseException,
        AttributeAlreadyDependentException, DependencyCycleException {
    // TODO: what if these don't have correct IDs?
    List<Integer> attributeIds = Evaluate.getVariableIds(equation);

    Attribute dependentAttribute = (Attribute) get(dependentAttributeId);
    if (dependentAttribute.isDependentAttribute()) {
        // it is already a dependent attribute
        throw new AttributeAlreadyDependentException();
    }

    if (!attributeIds.contains(dependentAttributeId)) { return null; }

    List<Attribute> attributes = new ArrayList<Attribute>();
    int spidrId = NULL_ID;
    for (Integer id : attributeIds) {
        Attribute a = (Attribute) get(id);
        if (!a.getClassName().equals(Attribute.class.getName())) { return null; }
        if (spidrId == NULL_ID) {
            spidrId = a.getSpidrId();
        }

        if ((a.getSpidrId() == NULL_ID) || (a.getSpidrId() != spidrId)) { return null; }
        attributes.add(a);
    }

    if (spidrId == NULL_ID) { return null; }

    Constraint constraint = new Constraint(name, equation,
            dependentAttributeId, getNextId(), spidrId);
    if (Constraint.containsDependencyCycles(constraint, this)) { throw new DependencyCycleException(); }

    // Update the database
    db.newRecord(currentTxn, constraint.writeKey(),
            constraint.writeClass(), constraint.writeEntry());
    for (Attribute att : attributes) {
        att.addConstraintId(constraint.getId());
        update(att);
    }
    dependentAttribute.propagateUpdate(this);

    SpidrObject so = (SpidrObject) get(spidrId);
    so.addConstraintId(constraint.getId());
    update(so);

    return constraint;
}

// Create a new Document
public Document createDocument(int parentFolderId, byte[] content,
        String name) throws DatabaseException {

    StateObject obj = get(parentFolderId);
    if (!(obj instanceof Folder)) { return null; }
    Folder folder = (Folder) obj;
    Document document = new Document(parentFolderId, content, name,
            getNextId(), folder.getSpidrId());
    folder.addDocumentId(document.getId());

    // Update the database
    db.newRecord(currentTxn, document.writeKey(), document.writeClass(),
            document.writeEntry());
    update(folder);

    return document;
}

// Create a new Edge
public Edge createEdge(int sourceNodeId, int targetNodeId, String name)
        throws DatabaseException {

    StateObject sObj = get(sourceNodeId);
    StateObject tObj = get(targetNodeId);

    if (!(sObj instanceof Node) || !(tObj instanceof Node)) { return null; }
    Node source = (Node) sObj;
    Node target = (Node) tObj;

    String longName = source.getName() + Edge.EDGE_DELIMITER + name
            + Edge.EDGE_DELIMITER + target.getName();

    // Make sure it isn't already present in the spidr
    if (parse(source.getSpidrId(), longName, false) != null) { return null; }

    Edge edge = new Edge(sourceNodeId, targetNodeId, longName, getNextId(),
            source.getSpidrId());
    source.addOutEdgeId(edge.getId());
    target.addInEdgeId(edge.getId());

    // Update the database
    db.newRecord(currentTxn, edge.writeKey(), edge.writeClass(), edge
            .writeEntry());
    update(source);
    update(target);
    createAttribute(edge.getId(), "Existence");

    return edge;
}

// Create a new Folder
public Folder createFolder(int parentId, String name)
        throws DatabaseException {

    StateObject obj = get(parentId);
    if (!(obj instanceof Folder)) {
        return null;
    } else {
        Folder parent = (Folder) obj;
        Folder folder = new Folder(parentId, name, getNextId(), parent
                .getSpidrId());
        parent.addFolderId(folder.getId());

        // Update the database
        db.newRecord(currentTxn, folder.writeKey(), folder.writeClass(),
                folder.writeEntry());
        update(parent);

        return folder;
    }
}

// Create a new Node
public Node createNode(int parentId, String name) throws DatabaseException {

    StateObject obj = get(parentId);
    if (!(obj instanceof Node)) { return null; }
    Node parent = (Node) obj;
    String longName = parent.getName() + Node.NODE_DELIMITER + name;

    // Make sure it isn't already present in the spidr
    if (parse(parent.getSpidrId(), longName, false) != null) { return null; }

    Node node = new Node(parentId, longName, getNextId(), parent
            .getSpidrId());
    parent.addChildId(node.getId());

    // Update the database
```

```
        db.newRecord(currentTxn, node.writeKey(), node.writeClass(), node
                .writeEntry());
        update(parent);

        // Inherit attributes from the parent
        for (Integer attributeId : parent.getAttributeIds()) {
            Attribute attribute = (Attribute) get(attributeId);
            createAttribute(node.getId(), attribute.getShortName());

        }

        return node;
    }

    // Create a new Quotation
    public Quotation createQuotation(int quotableObjectId, int documentId,
            int start, int end) throws DatabaseException {

        StateObject qobj = get(quotableObjectId);
        StateObject dobj = get(documentId);
        if (!(qobj instanceof QuotableObject) || !(dobj instanceof Document)) { return null; }
        QuotableObject qo = (QuotableObject) qobj;
        Document doc = (Document) dobj;
        Quotation q = new Quotation(quotableObjectId, documentId, start, end,
                getNextId(), qo.getSpidrId());
        qo.addQuotationId(q.getId());
        doc.addQuotationId(q.getId());

        // Update the database
        db.newRecord(currentTxn, q.writeKey(), q.writeClass(), q.writeEntry());
        update(qo);
        update(doc);

        return q;
    }

    // Create a new SpidrList
    public SpidrList createSpidrList() throws DatabaseException {
        SpidrList sl = new SpidrList(SPIDR_LIST_ID);

        db.newRecord(currentTxn, sl.writeKey(), sl.writeClass(), sl
                .writeEntry());

        return sl;
    }

    // Create a new "spidr"
    public SpidrObject createSpidrObject(String name) throws DatabaseException {
        int spidrId = getNextId();
        Node rootNode = createRootNode(spidrId);
        Folder rootFolder = createRootFolder(spidrId);
        SpidrObject so = new SpidrObject(rootNode.getId(), rootFolder.getId(),
                name, spidrId);

        SpidrList spidrList = (SpidrList) get(SPIDR_LIST_ID);
        spidrList.addSpidrId(so.getId());
        db.newRecord(currentTxn, so.writeKey(), so.writeClass(), so
                .writeEntry());
        db.updateRecord(currentTxn, spidrList.writeKey(), spidrList
                .writeEntry());
    return so;
}

// Delete the selected object
public boolean delete(int id) throws DatabaseException {
    StateObject obj = get(id);
    if (obj != null) {
        // Taken care of in SpidrObject.deleteMe()
        // if (obj instanceof SpidrObject) {
        // SpidrList spidrList = getSpidrList();
        // spidrList.removeSpidrId(id);
        // update(spidrList);
        // }
        obj.deleteMe(this);
        db.removeRecord(currentTxn, obj.writeKey());
        return true;
    }
    return false;
}

// Delete the selected object
public void delete(StateObject obj) throws DatabaseException {
    // Make sure we have fresh data
    delete(obj.getId());
}

// Updates the record for a StateObject, and returns a boolean
// indicating the success of the operation
public boolean update(StateObject obj) throws DatabaseException {
    return db.updateRecord(currentTxn, obj.writeKey(), obj.writeEntry())
            .equals(OperationStatus.SUCCESS);
}

// Transaction calls
public void beginTransaction() {
    if (currentTxn != null) { throw new RuntimeException(
            "There is still a transaction open: " + currentTxn); }
    try {
        currentTxn = environment.beginTransaction(null, null);
    } catch (DatabaseException e) {
        e.printStackTrace();
    }

}

public void commitTransaction() {
    try {
        currentTxn.commit();
    } catch (DatabaseException e) {
        e.printStackTrace();
    }
    currentTxn = null;
}

public void abortTransaction() {
    try {
        if (currentTxn != null) {
            currentTxn.abort();
        }
    } catch (DatabaseException e) {
        e.printStackTrace();
        }
        currentTxn = null;
    }
}
```

A.2 SpidrEnvironment.java

```
public class SpidrEnvironment {

    private Environment environment;

    private Spidr spidr;

    // The databases that our application uses
    private Database objectDb;
    private Database classDb;
    private SecondaryDatabase namesDb;

    // This database keeps track of the current id number
    private Database idDatabase;

    private DbBackup backupHelper;

    // The environment is not ready to use yet. setup() must be called first.
    public SpidrEnvironment() {}

    // The setup() method opens all our databases and the environment
    // for us.
    public void setup(File envHome, boolean readOnly) throws DatabaseException {

        if (!envHome.exists()) {
            envHome.mkdirs();
        }

        EnvironmentConfig envConfig = new EnvironmentConfig();
        DatabaseConfig dbConfig = new DatabaseConfig();
        SecondaryConfig namesConfig = new SecondaryConfig();
        namesConfig.setSortedDuplicates(true);
        namesConfig.setAllowPopulate(true);
        namesConfig.setKeyCreator(new NameKeyCreator());

        // If the environment is read-only, then
        // make the databases read-only too.
        envConfig.setReadOnly(readOnly);
        dbConfig.setReadOnly(readOnly);
        namesConfig.setReadOnly(readOnly);

        // If the environment is opened for write, then we want to be
        // able to create the environment and databases if
        // they do not exist.
        envConfig.setAllowCreate(!readOnly);
        dbConfig.setAllowCreate(!readOnly);
        namesConfig.setAllowCreate(!readOnly);

        envConfig.setTransactional(!readOnly);
        envConfig.setTxnTimeout(10000000); // 10 seconds
        dbConfig.setTransactional(!readOnly);
        namesConfig.setTransactional(!readOnly);
        // Open the environment
        environment = new Environment(envHome, envConfig);

        // Now open, or create and open, our databases
        // Open the vendors and inventory databases
        objectDb = environment.openDatabase(null, "ObjectDb", dbConfig);
        classDb = environment.openDatabase(null, "ClassDb", dbConfig);
        namesDb = environment.openSecondaryDatabase(null, "NamesDb", objectDb,
                namesConfig);

        idDatabase = environment.openDatabase(null, "IdDb", dbConfig);

        spidr = new Spidr(environment, objectDb, classDb, namesDb, idDatabase);

        backupHelper = new DbBackup(environment);
    }

    // getter methods

    // Needed for things like beginning transactions
    public Environment getEnvironment() {
        return environment;
    }

    public Database getDb() {
        return objectDb;
    }

    public Database getClassDb() {
        return classDb;
    }

    public Spidr getSpidr() {
        return spidr;
    }

    public void backup() throws DatabaseException {
        // Start backup, find out what needs to be copied.
        backupHelper.startBackup();
        try {
            String[] filesForBackup = backupHelper.getLogFilesInBackupSet();

            // Copy the files to archival storage.
            for (String filePath : filesForBackup) {
                File file = new File(environment.getHome().getPath() + "/"
                        + filePath);
                File backupDir = new File(environment.getHome().getPath()
                        + "/Backup/");
                if (!backupDir.exists()) {
                    backupDir.mkdirs();
                }
                File backup = new File(backupDir.getPath() + "/"
                        + file.getName());

                try {
                    copy(file, backup);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
    } finally {
        // Remember to exit backup mode, or all log files won't be cleaned
        // and disk usage will bloat.
        backupHelper.endBackup();
    }
}

private void copy(File src, File dst) throws IOException {
    InputStream in = new FileInputStream(src);
    OutputStream out = new FileOutputStream(dst);

    // Transfer bytes from in to out
    byte[] buf = new byte[1024];
    int len;
    while ((len = in.read(buf)) > 0) {
        out.write(buf, 0, len);
    }
    in.close();
    out.close();
}

// Close the environment
public void close() {
    if (environment != null) {

        // Close the secondary before closing the primaries
        if (namesDb != null) {
            try {
                namesDb.close();
            } catch (DatabaseException e) {
                System.err.println("closeEnv: myClassDb: " + e.toString());
                e.printStackTrace();
            }
        }

        if (objectDb != null) {
            try {
                objectDb.close();
            } catch (DatabaseException e) {
                System.err.println("closeEnv: myClassDb: " + e.toString());
                e.printStackTrace();
            }
        }

        if (classDb != null) {
            try {
                classDb.close();
            } catch (DatabaseException e) {
                System.err.println("closeEnv: myClassDb: " + e.toString());
                e.printStackTrace();
            }
        }

        if (idDatabase != null) {
            try {
                idDatabase.close();
            } catch (DatabaseException e) {
                System.err.println("closeEnv: myClassDb: " + e.toString());
                e.printStackTrace();
            }
        }
```

```
            try {
                environment.close();
            } catch (DatabaseException e) {
                System.err.println("closeEnv: myClassDb: " + e.toString());
                e.printStackTrace();
            }
        }
    }
}
```

A.3 SpidrDatabase.java

```
public class SpidrDatabase {
    private Database objectDb;
    private Database classDb;
    private SecondaryDatabase namesDb;

    public SpidrDatabase(Database objectDb, Database classDb,
            SecondaryDatabase namesDb) {
        this.objectDb = objectDb;
        this.classDb = classDb;
        this.namesDb = namesDb;
    }

    public OperationStatus newRecord(Transaction txn, DatabaseEntry key,
            DatabaseEntry classEntry, DatabaseEntry objEntry)
            throws DatabaseException {
        OperationStatus objStatus = objectDb.put(txn, key, objEntry);
        if (!objStatus.equals(OperationStatus.SUCCESS)) return objStatus;

        OperationStatus classStatus = classDb.put(txn, key, classEntry);
        return classStatus;
    }

    public OperationStatus updateRecord(Transaction txn, DatabaseEntry key,
            DatabaseEntry objEntry) throws DatabaseException {
        OperationStatus objStatus = objectDb.put(txn, key, objEntry);
        return objStatus;
    }

    public OperationStatus get(Transaction txn, DatabaseEntry key,
            DatabaseEntry classEntry, DatabaseEntry objEntry)
            throws DatabaseException {
        OperationStatus classStatus = classDb.get(txn, key, classEntry,
                LockMode.DEFAULT);
        if (!classStatus.equals(OperationStatus.SUCCESS)) return classStatus;
        return objectDb.get(txn, key, objEntry, LockMode.DEFAULT);
    }

    // Gets a list of all primary keys for the specified secondary key
    public List<DatabaseEntry> getInSecondary(Transaction txn,
            DatabaseEntry secKey) throws DatabaseException {

        SecondaryCursor cursor = namesDb.openSecondaryCursor(txn, null);

        DatabaseEntry primaryKey = new DatabaseEntry();
        OperationStatus nameStatus = cursor.getSearchKey(secKey, primaryKey,
                new DatabaseEntry(), LockMode.DEFAULT);
        List<DatabaseEntry> keys = new ArrayList<DatabaseEntry>();

        while (nameStatus.equals(OperationStatus.SUCCESS)) {
            keys.add(primaryKey);
            nameStatus = cursor.getNextDup(secKey, primaryKey,
                    new DatabaseEntry(), LockMode.DEFAULT);
        }

        cursor.close();
        return keys;
    }

    public OperationStatus removeRecord(Transaction txn, DatabaseEntry key)
            throws DatabaseException {
        OperationStatus classStatus = classDb.delete(txn, key);
        OperationStatus objStatus = objectDb.delete(txn, key);

        if (!classStatus.equals(OperationStatus.SUCCESS)) return classStatus;
        return objStatus;
    }
}
```

A.4 Attribute.java

```
public class Attribute extends QuotableObject {

    public static final String ATTRIBUTE_DELIMITER = ":";

    private int entityId; // The owning entity

    private List<Integer> attributeElementIds;
    private List<Integer> dynamicAEIds;
    private List<Integer> constraintIds;

    public Attribute() {
        this(-1, null, -1, -1);
    }

    public Attribute(int entityId, String name, int id, int spidrId) {
        super(name, id, spidrId);
        this.entityId = entityId;

        attributeElementIds = new ArrayList<Integer>();
        dynamicAEIds = null;
        constraintIds = new ArrayList<Integer>();
    }

    // Whether or not this attribute is the dependent attribute in some
    // constraint
    public boolean isDependentAttribute() {
        return (dynamicAEIds != null);
    }

    // Accessors
    public List<Integer> getAttributeElementIds() {
        if (dynamicAEIds == null) {
            // System.out.println(getId() + ": static");
        return attributeElementIds;
    } else {
        // System.out.println(getId() + ": dynamic");
        return dynamicAEIds;
    }
}

public List<Integer> getConstraintIds() {
    return constraintIds;
}

// Mutators
public void clearDynamicAEIds() {
    dynamicAEIds = null;
}

public void setDynamicAEIds(List<Integer> dynamicAEIds) {
    // System.out.println(getId() + " Setting dynamic AEIds: " +
    // dynamicAEIds);
    this.dynamicAEIds = dynamicAEIds;
}

public boolean addAttributeElementId(Integer id) {
    if (dynamicAEIds != null) { return false; }
    return attributeElementIds.add(id);
}

public boolean removeAttributeElementId(Integer id) {
    if (dynamicAEIds != null) { return false; }
    return attributeElementIds.remove(id);
}

protected void clearAttributeElementIds() {
    attributeElementIds.clear();
}

public boolean addConstraintId(Integer id) {
    return constraintIds.add(id);
}

public boolean removeConstraintId(Integer id) {
    return constraintIds.remove(id);
}

protected void clearConstraintIds() {
    constraintIds.clear();
}

@Override
public String getShortName() {
    String[] splitName = name.split(ATTRIBUTE_DELIMITER);
    // Return the name after the ":" char
    return splitName[splitName.length - 1];
}

public int getEntityId() {
    return entityId;
}

protected void setEntityId(int entityId) {
    this.entityId = entityId;
```

```
}

// Called when its elements have changed to propagate the change to any
// attributes that are dependent upon it
public void propagateUpdate(Spidr spidr) throws DatabaseException {
    Set<Constraint> affectedConstraints = new HashSet<Constraint>();
    findAffectedConstraints(affectedConstraints, spidr);
    Set<Integer> affectedAttributeIds = new HashSet<Integer>();
    for (Constraint c : affectedConstraints) {
        affectedAttributeIds.add(c.getDependentAttributeId());
    }

    while (!affectedConstraints.isEmpty()) {
        Iterator<Constraint> cIterator = affectedConstraints.iterator();
        while (cIterator.hasNext()) {
            Constraint c = cIterator.next();
            // Check if any of the independent attributes need to be
            // recalculated
            boolean indepAttributesValid = true;
            for (Integer attId : c.getIndependentAttributeIds()) {
                if (affectedAttributeIds.contains(attId)) {
                    // Can't calculate the value of the dep attribute yet,
                    // because the independent attributes haven't been
                    // updated yet
                    indepAttributesValid = false;
                    break;
                }
            }

            if (indepAttributesValid) {
                // All indep attributes are up-to-date: update the dependent
                // attribute
                c.recomputeAEs(spidr);
                cIterator.remove();
                affectedAttributeIds.remove(c.getDependentAttributeId());
            }
        }
    }
}

// Finds all constraints affected by changes in this attribute's elements
// Adds the found constraints to affectedConstraints
private void findAffectedConstraints(Set<Constraint> affectedConstraints,
        Spidr spidr) throws DatabaseException {
    for (Integer conId : getConstraintIds()) {
        Constraint c = (Constraint) spidr.get(conId);
        if (affectedConstraints.add(c)) {
            ((Attribute) spidr.get(c.getDependentAttributeId()))
                    .findAffectedConstraints(affectedConstraints, spidr);
        }
    }
}

@Override
public void deleteMe(Spidr spidr) throws DatabaseException {
    Entity entity = (Entity) spidr.get(getEntityId());
    entity.removeAttributeId(getId());
    spidr.update(entity);

    for (int objId : attributeElementIds) {
        spidr.delete(objId);
        }

        if (dynamicAEIds != null) {
            for (int objId : dynamicAEIds) {
                spidr.delete(objId);
            }
        }

        for (int objId : getConstraintIds()) {
            spidr.delete(objId);
        }

        super.deleteMe(spidr);
    }

    // Entry binding
    @Override
    public void readEntry(TupleInput in) {
        super.readEntry(in);

        setEntityId(in.readInt());

        int count = in.readInt();
        clearAttributeElementIds();
        for (int i = 0; i < count; i++) {
            addAttributeElementId(in.readInt());
        }

        try {
            count = in.readInt();
        } catch (IndexOutOfBoundsException e) {
            // We must be using an old version of the DB
            // System.out.println("Read old attribute data");
            return;
        }
        clearDynamicAEIds();
        if (count != -1) {
            dynamicAEIds = new ArrayList<Integer>();
            for (int i = 0; i < count; i++) {
                dynamicAEIds.add(in.readInt());
            }
        }

        count = in.readInt();
        clearConstraintIds();
        for (int i = 0; i < count; i++) {
            addConstraintId(in.readInt());
        }
    }

    @Override
    public void writeEntry(TupleOutput out) {
        super.writeEntry(out);

        out.writeInt(getEntityId());

        List<Integer> ids = attributeElementIds;
        out.writeInt(ids.size());
        for (Integer id : ids) {
            out.writeInt(id);
        }
        ids = dynamicAEIds;
        if (ids == null) {
            out.writeInt(-1);
        } else {
            out.writeInt(ids.size());
            for (Integer id : ids) {
                out.writeInt(id);
            }
        }

        ids = getConstraintIds();
        out.writeInt(ids.size());
        for (Integer id : ids) {
            out.writeInt(id);
        }
    }

    @Override
    public boolean sameData(Object obj) {
        if (!(obj instanceof Attribute) || !super.sameData(obj)) return false;

        Attribute attribute = (Attribute) obj;
        return (getEntityId() == attribute.getEntityId()
                && getAttributeElementIds().equals(
                        attribute.getAttributeElementIds()) && getConstraintIds()
                .equals(attribute.getConstraintIds()));
    }

    @Override
    public int hashCode() {
        return (dynamicAEIds == null ? 5 : dynamicAEIds.hashCode())
                + 17
                * (entityId + 97 * (attributeElementIds.hashCode() + 29 * (constraintIds
                        .hashCode() + 31 * super.hashCode())));
    }

    public Map<String, Object> getData(String entityClassName) {
        Map<String, Object> data = super.getData();

        data.put(entityClassName, entityId);
        data.put("elements", getAttributeElementIds().toArray());
        data.put("constraints", getConstraintIds().toArray());
        data.put("class", "attribute");
        return data;
    }

    @Override
    public Map<String, Object> getData() {
        return getData("entity");
    }
}
```

A.5 Constraint.java

```
public class Constraint extends QuotableObject {

    // The defining equation
    private String equation;
    private Integer dependentAttributeId;

    // NOTE: These constructors create java objects but do NOT
    // create database objects. See Spidr.createConstraint()
    public Constraint() {
        this(null, null, -1, -1, -1);
    }

    public Constraint(String name, String equation, int dependentAttributeId,
            int id, int spidrId) {
        super(name, id, spidrId);

        this.equation = equation;
        this.dependentAttributeId = dependentAttributeId;
    }

    // Accessors
    public List<Integer> getAttributeIds() {
        return Evaluate.getVariableIds(getEquation());
    }

    public List<Integer> getIndependentAttributeIds() {
        List<Integer> ids = Evaluate.getVariableIds(getEquation());
        ids.remove(getDependentAttributeId());
        return ids;
    }

    public String getEquation() {
        return equation;
    }

    public Integer getDependentAttributeId() {
        return dependentAttributeId;
    }

    // Mutators
    protected void setEquation(String equation) {
        this.equation = equation;
    }

    public void setDependentAttributeId(Integer id) {
        if (getAttributeIds().contains(id)) {
            dependentAttributeId = id;
        }
    }

    // Tests whether there are dependency cycles involving the given constraint
    public static boolean containsDependencyCycles(Constraint constraint,
            Spidr spidr) throws DatabaseException {
        Attribute originalDependent = (Attribute) spidr.get(constraint
                .getDependentAttributeId());
        Set<Attribute> visited = new HashSet<Attribute>();
        visited.add(originalDependent);
        return containsDependencyCycles(constraint, spidr, visited,
                originalDependent);
```

```
}

// This is a depth-first search with a visited list
private static boolean containsDependencyCycles(Constraint constraint,
        Spidr spidr, Set<Attribute> visited, Attribute originalDependent)
        throws DatabaseException {
    for (Integer attId : constraint.getIndependentAttributeIds()) {
        Attribute a = (Attribute) spidr.get(attId);

        if (!visited.contains(a)) {
            if (a.equals(originalDependent)) {
                // We looped back through dependencies to the original
                // attribute!
                return true;
            }
            visited.add(a);
            // Continue the search through the indep attribute a
            for (Integer conId : a.getConstraintIds()) {
                if (containsDependencyCycles((Constraint) spidr.get(conId),
                        spidr, visited, originalDependent)) { return true; }
            }
        }
    }

    return false;
}

// Recomputes the AEs for the dependent attribute
public void recomputeAEs(Spidr spidr) throws DatabaseException {
    List<Integer> attributeIds = Evaluate.getVariableIds(getEquation());
    attributeIds.remove(getDependentAttributeId());

    Set<List<AttributeElement>> overlaps = findOverlaps(attributeIds, spidr);

    List<Integer> dynamicAEIds = new ArrayList<Integer>();
    for (List<AttributeElement> overlap : overlaps) {
        if (overlap.size() != 0) {
            Map<String, Double> bindings = new HashMap<String, Double>();
            for (AttributeElement elt : overlap.subList(1, overlap.size())) {
                bindings.put(String.valueOf(elt.getAttributeId()), Double
                        .valueOf(elt.getValue()));
            }

            double value;
            try {
                value = Evaluate.evaluate(equation, bindings);
                if (value != 1.0) { throw new RuntimeException(
                        "Poorly formed equation"); }
                AttributeElement ae = spidr
                        .createAttributeElement(getDependentAttributeId(),
                                String.valueOf(bindings
                                        .get(getDependentAttributeId()
                                                .toString())), overlap.get(
                                        0).getStartTime(), overlap.get(0)
                                        .getEndTime(), true);
                dynamicAEIds.add(ae.getId());
            } catch (UnboundVariableException e) {
                throw new RuntimeException("Poorly formed equation");
            }
        }
    }
    Attribute dependentAttribute = (Attribute) spidr
            .get(getDependentAttributeId());
    dependentAttribute.setDynamicAEIds(dynamicAEIds);
    spidr.update(dependentAttribute);
}

// Finds all overlaps of the AE periods for a set of attributes,
// and returns the "paths" of AEs for those overlapping periods.
// At the start of each "path" is an AE storing the start and end dates.
private static Set<List<AttributeElement>> findOverlaps(
        List<Integer> attributeIds, Spidr spidr) throws DatabaseException {
    List<List<AttributeElement>> aeLists = new ArrayList<List<AttributeElement>>();

    for (Integer id : attributeIds) {
        Attribute attribute = (Attribute) spidr.get(id);
        List<AttributeElement> aeList = new ArrayList<AttributeElement>();
        for (Integer aeid : attribute.getAttributeElementIds()) {
            AttributeElement ae = (AttributeElement) spidr.get(aeid);
            aeList.add(ae);
        }
        aeLists.add(aeList);
    }

    return findOverlaps(aeLists);
}

// Finds all overlaps of the AE periods for a set of attributes,
// and returns the "paths" of AEs for those overlapping periods.
// At the start of each "path" is an AE storing the start and end dates.
private static Set<List<AttributeElement>> findOverlaps(
        List<List<AttributeElement>> attributes) {

    Set<List<AttributeElement>> results = new HashSet<List<AttributeElement>>();

    if (attributes.size() == 0) {
        return results;
    } else if (attributes.size() == 1) {
        for (AttributeElement ae : attributes.get(0)) {
            List<AttributeElement> path = new ArrayList<AttributeElement>();
            path.add(ae);
            path.add(ae);
            results.add(path);
        }
        return results;
    }

    // Recursively find sub-paths
    Set<List<AttributeElement>> paths = findOverlaps(attributes.subList(1,
            attributes.size()));
    for (List<AttributeElement> path : paths) {
        for (AttributeElement ae : attributes.get(0)) {
            AttributeElement overlap = ae.findOverlap(path.get(0));
            if (overlap != null) {
                List<AttributeElement> extendedPath = new ArrayList<AttributeElement>(
                        path.subList(1, path.size()));
                extendedPath.add(0, overlap);
                extendedPath.add(1, ae);
                results.add(extendedPath);
            }
        }
    }

    return results;
}

@Override
public void deleteMe(Spidr spidr) throws DatabaseException {
    for (int objId : getAttributeIds()) {
        Attribute attribute = (Attribute) spidr.get(objId);
        attribute.removeConstraintId(getId());
        if (attribute.getId() == getDependentAttributeId()) {
            for (Integer id : attribute.getAttributeElementIds()) {
                spidr.delete(id);
            }
            attribute.clearDynamicAEIds();
            attribute.propagateUpdate(spidr);
        }

        spidr.update(attribute);

        SpidrObject so = (SpidrObject) spidr.get(getSpidrId());
        so.removeConstraintId(getId());
        spidr.update(so);
    }

    super.deleteMe(spidr);
}

// Entry binding
@Override
public void readEntry(TupleInput in) {
    super.readEntry(in);

    setEquation(in.readString());
    setDependentAttributeId(in.readInt());
}

@Override
public void writeEntry(TupleOutput out) {
    super.writeEntry(out);

    out.writeString(getEquation());
    out.writeInt(getDependentAttributeId());
}

@Override
public boolean sameData(Object obj) {
    if (!(obj instanceof Constraint) || !super.sameData(obj)) return false;

    Constraint constraint = (Constraint) obj;
    return (getEquation().equals(constraint.getEquation()) && getDependentAttributeId()
            .equals(constraint.getDependentAttributeId()));
}

@Override
public int hashCode() {
    return equation.hashCode() + 97
            * (dependentAttributeId + 31 * super.hashCode());
}

@Override
```

```
    public String prettyPrint() {
        return super.prettyPrint() + "\n Equation: " + getEquation()
                + "\n Dependent variable ID: " + getDependentAttributeId();
    }

    @Override
    public String toString() {
        return "Equation: " + getEquation() + ", dependent: "
                + getDependentAttributeId();
    }

    @Override
    public Map<String, Object> getData() {
        Map<String, Object> data = super.getData();

        data.put("equation", getEquation());
        data.put("attributes", getAttributeIds().toArray());
        data.put("dependent", getDependentAttributeId());
        data.put("class", "constraint");
        return data;
    }
}
```

Appendix B

Selected user interface source code

B.1 ConstraintPanel.java

```
public class ConstraintPanel extends JPanel {

    /**
     *
     */
    private static final long serialVersionUID = 2759243521723217207L;
    private JList constraintList;
    private DefaultListModel constraintModel;
    private Spidr spidr;

    public ConstraintPanel(List<Constraint> constraints, Spidr spidr) {

        this.spidr = spidr;
        setupUI();

        for (Constraint c : constraints) {
            addConstraint(c);
        }

    }

    private void setupUI() {
        constraintModel = new DefaultListModel();
        constraintList = new JList(constraintModel);
        constraintList.setCellRenderer(new ConstraintListCellRenderer());
        JScrollPane scrollPane = new JScrollPane(constraintList);
        scrollPane.setPreferredSize(new Dimension(400, 300));
        setLayout(new BorderLayout());
        add(scrollPane, BorderLayout.CENTER);
        JButton newButton = new JButton("New constraint");
        newButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                final CreateConstraintDialog dialog = new CreateConstraintDialog(
                        spidr, ConstraintPanel.this);
                dialog.addListener(new CreateDialogListener() {
                    public void fireCreated() {
                        Constraint constraint = dialog.getConstraint();
                    if (!constraintModel.contains(constraint)) {
                        addConstraint(constraint);
                    }
                }
            });
            dialog.setVisible(true);
        }
    });
    add(newButton, BorderLayout.PAGE_END);

    ConstraintPanelEventAdapter cpea = new ConstraintPanelEventAdapter();
    constraintList.addKeyListener(cpea);
    constraintList.addMouseListener(cpea);
}

private void addConstraint(Constraint c) {
    constraintModel.addElement(c);
}

private void removeConstraint(Constraint c) {
    constraintModel.removeElement(c);
}

@SuppressWarnings("unused")
private void clearConstraints() {
    constraintModel.clear();
}

private void openSelected() {
    Constraint selected = (Constraint) constraintList.getSelectedValue();

    if (selected != null) {
        CreateConstraintDialog dialog = new CreateConstraintDialog(
                selected, spidr, this);
        dialog.setVisible(true);
    }
}

private void deleteSelected() {
    Constraint selected = (Constraint) constraintList.getSelectedValue();

    if (selected != null) {
        try {
            selected.remove();
            removeConstraint(selected);
        } catch (ClientConnectionException e) {
            e.printStackTrace();
        } catch (ObjectDeletedException e) {
            e.printStackTrace();
        }
    }
}

private class ConstraintPanelEventAdapter implements MouseListener,
        KeyListener {

    public void mouseClicked(MouseEvent evt) {
        if (evt.getClickCount() == 2) {
            openSelected();
        }
    }
```

```
        public void mouseEntered(MouseEvent evt) {}

        public void mouseExited(MouseEvent evt) {}

        public void mousePressed(MouseEvent evt) {}

        public void mouseReleased(MouseEvent evt) {}

        public void keyPressed(KeyEvent ke) {
            switch (ke.getKeyCode()) {
            case KeyEvent.VK_DELETE:
                deleteSelected();
                break;
            case KeyEvent.VK_ENTER:
                openSelected();
            }
        }

        public void keyReleased(KeyEvent ke) {}

        public void keyTyped(KeyEvent ke) {}
    }
}
```

B.2 ConstraintListCellRenderer.java

```
public class ConstraintListCellRenderer extends DefaultListCellRenderer {

    /**
     *
     */
    private static final long serialVersionUID = 8669439615350835291L;

    @Override
    public Component getListCellRendererComponent(JList list, Object value,
            int index, boolean isSelected, boolean hasFocus) {

        super.getListCellRendererComponent(list, value, index, isSelected,
                hasFocus);

        if (value instanceof Constraint) {
            Constraint c = (Constraint) value;
            try {
                setText(c.getEquation());
                Attribute dependent = c.getDependentAttribute();
                String indString = "";
                for (Attribute a : c.getAttributes()) {
                    if (a.getId() != dependent.getId()) {
                        indString += a.getName();
                        indString += ", ";
                    }
                }
                if (indString.length() > 0) {
                    indString = indString.substring(0, indString.length() - 2);
                }
                setToolTipText("<html>Eqn: " + c.getEquation() + "<br/>Dep: "
                        + dependent.getName() + "<br/>Ind: " + indString
                    + "</html>");
            } catch (ObjectDeletedException e) {
                e.printStackTrace();
            } catch (ClientConnectionException e) {
                e.printStackTrace();
            }
        }

        return this;
    }
}
```

B.3 CreateConstraintDialog.java

```
public class CreateConstraintDialog extends CreateDialog {

    /**
     *
     */
    private static final long serialVersionUID = -2046902219187980048L;
    private static final int INVALID_EQUATION = -2;
    private static final int FIELDS_VALID = 1;

    private EquationTextPane equationField;
    private JComboBox dependentList;
    // The constraint
    private Constraint constraint;

    private Spidr spidr;

    public CreateConstraintDialog(Constraint constraint, Spidr spidr,
            Component parentComponent) {

        super(parentComponent);

        this.constraint = constraint;
        this.spidr = spidr;

        setupUI();
    }

    public CreateConstraintDialog(Spidr spidr, Component parentComponent) {
        this(null, spidr, parentComponent);
    }

    private void setupUI() {
        setTitle("Define constraint");

        final JButton createButton = new JButton((constraint == null)
                ? "Create" : "Update");
        createButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                doCreate();
            }
        });
        final JButton cancelButton = new JButton("Cancel");
        cancelButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
```

```
        doCancel();
    }
});

// Disable the tab key
Set<KeyStroke> keyStrokes = new HashSet<KeyStroke>();
keyStrokes.add(KeyStroke.getKeyStroke(KeyEvent.VK_TAB,
        KeyEvent.CTRL_DOWN_MASK));
setFocusTraversalKeys(KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS,
        keyStrokes);

dependentList = new JComboBox();

cancelButton.setMnemonic(KeyEvent.VK_ESCAPE);
JPanel bottomButtonPanel = new JPanel();
bottomButtonPanel.setLayout(new FlowLayout());
bottomButtonPanel.add(createButton);
bottomButtonPanel.add(cancelButton);
bottomButtonPanel.setBorder(new EmptyBorder(5, 5, 0, 5));

JPanel mainPanel = new JPanel();
mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.PAGE_AXIS));
equationField = new EquationTextPane(spidr);
equationField.setPreferredSize(new Dimension(150, 100));
equationField.getDocument().addDocumentListener(new DocumentListener() {
    public void changedUpdate(DocumentEvent e) {
        // Repopulate the list
        if (constraint == null) {
            try {
                populateDependentList();
            } catch (ObjectDeletedException e1) {
                e1.printStackTrace();
            } catch (ClientConnectionException e1) {
                e1.printStackTrace();
            }
        }
    }

    public void insertUpdate(DocumentEvent e) {}

    public void removeUpdate(DocumentEvent e) {}
});

JScrollPane equationScrollPane = new JScrollPane(equationField);
mainPanel.add(equationScrollPane/* , BorderLayout.CENTER */);
JPanel labelPanel = new JPanel();
labelPanel.setLayout(new BoxLayout(labelPanel, BoxLayout.LINE_AXIS));
labelPanel.add(Box.createRigidArea(new Dimension(3, 1)));
labelPanel.add(new JLabel("Dependent attribute:"));
labelPanel.add(Box.createHorizontalGlue());
mainPanel.add(labelPanel);
mainPanel.add(dependentList);
mainPanel.add(bottomButtonPanel);
add(mainPanel);

if (constraint != null) {
    try {
        equationField.setText(constraint.getEquation());
        equationField.setEditable(false);
        populateDependentList();
    } catch (ClientConnectionException e) {
            JOptionPane.showMessageDialog(this,
                    "There was a connection error", "Connection error",
                    JOptionPane.ERROR_MESSAGE);
        } catch (ObjectDeletedException e) {
            JOptionPane.showMessageDialog(this,
                    "Error: object has been deleted", "Object deleted",
                    JOptionPane.ERROR_MESSAGE);
        }
    }

    // Handle window closing correctly.
    setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
    addWindowListener(new WindowAdapter() {
        @Override
        public void windowClosing(WindowEvent we) {
            doCancel();
        }
    });

    // Set the create button as the default
    getRootPane().setDefaultButton(createButton);

    // This is so the dialog will close with the ESC key
    KeyStroke escape = KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0, false);
    Action escapeAction = new AbstractAction() {
        private static final long serialVersionUID = 1L;

        public void actionPerformed(ActionEvent e) {
            doCancel();
        }
    };
    getRootPane().getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW).put(
            escape, "ESCAPE");
    getRootPane().getActionMap().put("ESCAPE", escapeAction);
    setPreferredSize(new Dimension(300, 200));
    pack();
    equationField.requestFocusInWindow();
}

public Constraint getConstraint() {
    return constraint;
}

private void populateDependentList() throws ObjectDeletedException,
        ClientConnectionException {
    if (constraint != null) {
        dependentList.removeAllItems();
        for (String aName : Constraint.getAttributeNames(equationField
                .getText())) {
            dependentList.addItem(aName);
        }
        dependentList.setSelectedItem(EquationTextPane
                .removeLeadingNode(constraint.getDependentAttribute()
                        .getName()));
    } else {
        Object selectedItem = dependentList.getSelectedItem();
        dependentList.removeAllItems();
        for (String aName : Constraint.getAttributeNames(equationField
                .getText())) {
            dependentList.addItem(aName);
        }
```

```
            dependentList.setSelectedItem(selectedItem);
        }
    }

    private void doCreate() {
        if (validateEquation() != 1) {
            JOptionPane.showMessageDialog(this,
                    "Constraint could not be created due to invalid inputs",
                    "Invalid inputs", JOptionPane.ERROR_MESSAGE);
            return;
        }

        try {
            String daName = (String) dependentList.getSelectedItem();

            Attribute dependentAttribute = (daName == null) ? null
                    : (Attribute) spidr.parse(daName, false);

            if (dependentAttribute == null) {
                JOptionPane.showMessageDialog(this,
                        "Constraint could not be created because "
                                + "the dependent attribute "
                                + "was not properly specified",
                        "Invalid input", JOptionPane.ERROR_MESSAGE);
                return;
            }

            for (Constraint c : dependentAttribute.getConstraints()) {
                if (c.getDependentAttribute().getId().equals(
                        dependentAttribute.getId())) {
                    JOptionPane.showMessageDialog(this, dependentAttribute
                            .getName()
                            + " could not be set as dependent"
                            + " because it is already\n"
                            + "the dependent attribute in the "
                            + "constraint shown below.\nEquation: "
                            + c.getEquation(), "Invalid input",
                            JOptionPane.ERROR_MESSAGE);
                    return;
                }
            }

            if (constraint == null) {
                constraint = spidr.makeConstraint(equationField.getText(),
                        dependentAttribute);
                if (constraint.getId() == frog.server.Spidr.ATT_ALREADY_DEP_ERROR_ID) {
                    JOptionPane.showMessageDialog(this,
                            "The constraint could not be created because\n"
                                    + dependentAttribute.getName()
                                    + "\nis already the dependent"
                                    + " attribute in another constraint.",
                            "Invalid input", JOptionPane.ERROR_MESSAGE);
                    return;
                } else if (constraint.getId() == frog.server.Spidr.CYCLE_ERROR_ID) {
                    JOptionPane.showMessageDialog(this,
                            "The constraint could not be created because"
                                    + " it would cause a dependency cycle.\n"
                                    + "Changing the dependent attribute"
                                    + " may prevent this problem.",
                            "Invalid input", JOptionPane.ERROR_MESSAGE);
                    return;
                }
            } else {
                boolean success = constraint
                        .setDependentAttribute(dependentAttribute);
                if (!success) {
                    JOptionPane.showMessageDialog(this, dependentAttribute
                            .getName()
                            + " could not be set as dependent.\n"
                            + "It is likely this is because doing"
                            + " so would cause a dependency loop.",
                            "Invalid input", JOptionPane.ERROR_MESSAGE);
                    return;
                }
            }
            setVisible(false);
            fireCreated();
            dispose();
        } catch (ClientConnectionException e) {
            JOptionPane.showMessageDialog(this, "There was a connection error",
                    "Connection error", JOptionPane.ERROR_MESSAGE);
        } catch (ObjectDeletedException e) {
            JOptionPane.showMessageDialog(this,
                    "Error: object has been deleted", "Object deleted",
                    JOptionPane.ERROR_MESSAGE);
        }
    }

    private void doCancel() {
        // user closed dialog or clicked cancel
        dispose();
    }

    // Performs validation on the equation field
    private int validateEquation() {
        // Check parentheses
        int openP = 0; // number of unmatched '(' so far
        int openB = 0; // number of unmatched '[' so far
        for (char c : equationField.getText().toCharArray()) {
            if (c == EquationTextPane.OPEN_PAREN.charAt(0)) {
                openP++;
            } else if (c == EquationTextPane.CLOSE_PAREN.charAt(0)) {
                openP--;
                if (openP < 0) {
                    // too many ')' already
                    return INVALID_EQUATION;
                }
            } else if (c == EquationTextPane.OPEN_BRACE.charAt(0)) {
                openB++;
            } else if (c == EquationTextPane.CLOSE_BRACE.charAt(0)) {
                openB--;
                if (openB < 0) {
                    // too many ']' already
                    return INVALID_EQUATION;
                }
            }
        }
        if ((openP != 0) || (openB != 0)) { return INVALID_EQUATION; }

        return FIELDS_VALID;
    }
}
```

B.4 EquationTextPane.java

```
public class EquationTextPane extends JTextPane {

    /**
     *
     */
    private static final long serialVersionUID = -6626957888453100905L;

    public static final int AUTOCOMPLETE_KEY = KeyEvent.VK_TAB;
    public static final String OPEN_BRACE = "[";
    public static final String CLOSE_BRACE = "]";
    public static final String OPEN_PAREN = "(";
    public static final String CLOSE_PAREN = ")";

    private Spidr spidr;
    private int autoStart, autoLen; // Start index and length of autocomplete
    // Whether or not we are editing autocomplete text
    private boolean editingAutoComplete = false;
    private StringBuffer oldDocText = new StringBuffer();

    public EquationTextPane(Spidr spidr) {
        super();
        this.spidr = spidr;
        autoStart = -1;
        autoLen = 0;
        StyledDocument doc = getStyledDocument();
        addStylesToDocument(doc);
        doc.setLogicalStyle(0, doc.getStyle("regular"));

        doc.addDocumentListener(new DocumentListener() {

            public void changedUpdate(DocumentEvent evt) {}

            public void insertUpdate(final DocumentEvent evt) {
                try {
                    final int offset = evt.getOffset();
                    final Document doc = getDocument();
                    final String insertText = doc.getText(offset, evt
                            .getLength());

                    // Update the old text
                    oldDocText.insert(offset, insertText);
                    if (!editingAutoComplete) {
                        SwingUtilities.invokeLater(new Runnable() {
                            public void run() {
                                try {
                                    if (evt.getLength() == 1) {
                                        if (insertText.equals(OPEN_BRACE)) {
                                            // Auto-insert closing brace
                                            editingAutoComplete = true;
                                            int oldCaretPos = getCaretPosition();
                                            doc.insertString(offset + 1,
                                                    CLOSE_BRACE, null);
                                            setCaretPosition(oldCaretPos);
                                            editingAutoComplete = false;
                            } else if (insertText
                                    .equals(OPEN_PAREN)) {
                                // Auto-insert closing paren
                                editingAutoComplete = true;
                                int oldCaretPos = getCaretPosition();
                                doc.insertString(offset + 1,
                                        CLOSE_PAREN, null);
                                setCaretPosition(oldCaretPos);
                                editingAutoComplete = false;
                            } else if (insertText
                                    .equals(CLOSE_BRACE)
                                    && (offset != 0)
                                    && doc.getText(offset - 1, 1)
                                            .equals(OPEN_BRACE)) {
                                // Ignore
                                editingAutoComplete = true;
                                doc.remove(offset + 1, 1);
                                editingAutoComplete = false;
                            } else if (insertText
                                    .equals(CLOSE_PAREN)
                                    && (offset != 0)
                                    && doc.getText(offset - 1, 1)
                                            .equals(OPEN_PAREN)) {
                                // Ignore
                                editingAutoComplete = true;
                                doc.remove(offset + 1, 1);
                                editingAutoComplete = false;
                            }
                        }
                    } catch (BadLocationException e) {
                        // Should not happen
                        e.printStackTrace();
                    }
                }
            });
        }
    } catch (BadLocationException e1) {
        // Should not happen
        e1.printStackTrace();
    }
}

public void removeUpdate(final DocumentEvent evt) {
    final int offset = evt.getOffset();
    final int len = evt.getLength();

    // Auto-remove closing brace
    if (!editingAutoComplete) {
        final Document doc = getDocument();
        if (len == 1) {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    editingAutoComplete = true;

                    if ((oldDocText.charAt(offset) == OPEN_BRACE
                            .charAt(0))
                            && (oldDocText.length() > offset + 1)
                            && (oldDocText.charAt(offset + 1) == (CLOSE_BRACE
                                    .charAt(0)))) {
                        try {
                            doc.remove(offset, 1);
```

```
                                } catch (BadLocationException e) {
                                    // Should never happen
                                    e.printStackTrace();
                                }
                            } else if ((oldDocText.charAt(offset) == OPEN_PAREN
                                    .charAt(0))
                                    && (oldDocText.length() > offset + 1)
                                    && (oldDocText.charAt(offset + 1) == (CLOSE_PAREN
                                            .charAt(0)))) {
                                try {
                                    doc.remove(offset, 1);
                                } catch (BadLocationException e) {
                                    // Should never happen
                                    e.printStackTrace();
                                }
                            }

                            editingAutoComplete = false;
                            // Update the old text
                            oldDocText.delete(offset, offset + len);
                        }
                    });
                } else {
                    // Update the old text
                    oldDocText.delete(offset, offset + len);
                }
            } else {
                // Update the old text
                oldDocText.delete(offset, offset + len);
            }
        }
    });

    addKeyListener(new KeyAdapter() {
        @Override
        public void keyPressed(KeyEvent e) {
            switch (e.getKeyCode()) {
            case AUTOCOMPLETE_KEY:
                autoComplete();
                e.consume();
                return;
            default:
                break;
            }
        }
    });

    addCaretListener(new CaretListener() {

        public void caretUpdate(CaretEvent e) {
            if (!editingAutoComplete) {
                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        updateAutoComplete();
                    }
                });
            }
        }
    });
}
```

```
// Updates the autocompletion
private void updateAutoComplete() {
    editingAutoComplete = true;
    int startIndex = findEntryStartIndex();
    int endIndex = findEntryEndIndex();
    int caretPos = getCaretPosition();
    if (startIndex != -1 && endIndex != -1) {
        autoStart = caretPos;
        autoLen = endIndex - caretPos;
        clearAutoCompletion();
        try {
            // System.out.println(startIndex + " " + caretPos);
            showAutoCompletion(getDocument().getText(startIndex,
                    caretPos - startIndex), caretPos);
        } catch (BadLocationException e) {
            // Should never happen
            e.printStackTrace();
        }
    } else {
        clearAutoCompletion();
    }

    setStyles();
    editingAutoComplete = false;
}

// Finds the start of an entry, denoted by OPEN_BRACE
// Returns starting index, or -1 if not found
private int findEntryStartIndex() {
    try {
        int caretPos = getCaretPosition();
        String leadingText = getDocument().getText(0, caretPos);
        // Find nearest occurring OPEN_BRACE and CLOSE_BRACE
        int openIndex = leadingText.lastIndexOf(OPEN_BRACE);
        int closeIndex = leadingText.lastIndexOf(CLOSE_BRACE);
        if ((openIndex != -1)
                && ((closeIndex == -1) || (openIndex > closeIndex))) { return openIndex + 1; }
    } catch (BadLocationException e) {
        // Will never happen
    }

    return -1;
}

// Finds the end of an entry, denoted by CLOSE_BRACE
// Returns index of the closing CLOSE_BRACE, or -1 if not found
private int findEntryEndIndex() {
    try {
        int caretPos = getCaretPosition();
        String trailingText = getDocument().getText(caretPos,
                getDocument().getLength() - caretPos);
        // Find nearest occurring OPEN_BRACE and CLOSE_BRACE
        int openIndex = trailingText.indexOf(OPEN_BRACE);
        int closeIndex = trailingText.indexOf(CLOSE_BRACE);
        if ((closeIndex != -1)
                && ((openIndex == -1) || (closeIndex < openIndex))) { return closeIndex
                + caretPos; }
    } catch (BadLocationException e) {
        // Will never happen
        e.printStackTrace();
    }
    return -1;
}

// Automatically sets styles throughout the document based on what part is
// being autocompleted
private void setStyles() {
    StyledDocument doc = getStyledDocument();
    doc.setCharacterAttributes(0, doc.getLength(), doc.getStyle("regular"),
            true);
    if (autoStart != -1) {
        doc.setCharacterAttributes(autoStart, autoLen,
                doc.getStyle("bold"), true);
    }
}

private void autoComplete() {
    int endIndex = findEntryEndIndex();
    if ((findEntryStartIndex() != -1) && (endIndex != -1)) {
        autoStart = -1;
        if (getCaretPosition() == endIndex) {
            setCaretPosition(endIndex + 1);
        } else {
            setCaretPosition(endIndex);
        }
    }
}

private void clearAutoCompletion() {
    if (autoStart != -1) {
        try {
            getStyledDocument().remove(autoStart, autoLen);
            // System.out.println(" " + getText());
            autoStart = -1;
            autoLen = 0;
        } catch (BadLocationException e) {
            e.printStackTrace();
        }
    }
}

private void showAutoCompletion(String searchString, int insertPos) {
    try {
        String autoCompleteText = "";
        StyledDocument doc = getStyledDocument();
        if (!searchString.equals("")) {
            List<DatabaseObject> sObjects = spidr.query(searchString);
            if (sObjects.size() != 0) {
                autoCompleteText = removeLeadingNode(
                        sObjects.get(0).getName()).substring(
                        searchString.length());
            }
        }

        doc.insertString(insertPos, autoCompleteText, null);
        setCaretPosition(insertPos);
        autoStart = insertPos;
        autoLen = autoCompleteText.length();
    } catch (ClientConnectionException e) {
        JOptionPane.showMessageDialog(this, "There was a connection error",
                "Connection error", JOptionPane.ERROR_MESSAGE);
        } catch (BadLocationException e) {
            // Should never happen
        } catch (ObjectDeletedException e) {
            JOptionPane.showMessageDialog(this,
                    "Error: object has been deleted", "Object deleted",
                    JOptionPane.ERROR_MESSAGE);
        }
    }

    public static String removeLeadingNode(String s) {
        if (s.indexOf("Node.") != 0) { return s; }
        return s.substring(5);
    }

    protected void addStylesToDocument(StyledDocument doc) {
        // Initialize some styles.
        Style def = StyleContext.getDefaultStyleContext().getStyle(
                StyleContext.DEFAULT_STYLE);

        Style regular = doc.addStyle("regular", def);
        StyleConstants.setFontFamily(def, "SansSerif");
        StyleConstants.setFontSize(regular, 12);

        Style s = doc.addStyle("italic", regular);
        StyleConstants.setItalic(s, true);

        s = doc.addStyle("bold", regular);
        StyleConstants.setBold(s, true);

        s = doc.addStyle("small", regular);
        StyleConstants.setFontSize(s, 10);

        s = doc.addStyle("large", regular);
        StyleConstants.setFontSize(s, 16);
    }
}
```

Bibliography

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

[2] P. J. Clarkson, C. Simons, and C. Eckert. Predicting change propagation in complex design. *ASME 2001 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2001.

[3] Oracle Corporation. Oracle Berkeley DB Java Edition. `http://www.oracle.com/database/berkeley-db/je/index.html`.

[4] Apache Software Foundation. ws-xmlrpc - Apache XML-RPC. `http://ws.apache.org/xmlrpc/`.

[5] Franz Inc. AllegroGraph. `http://agraph.franz.com/allegrograph/`.

[6] J. G. Miller. *Living Systems.* McGraw-Hill, New York, 1978.

[7] Igor A. Sylvester. A Hierarchical Systems Knowledge Representation Framework. Master's thesis, Massachusetts Institute of Technology, 2007.