An Artifact Modeling Approach for Developing Integrated Engineering Systems

Sivand Lakmazaheri *

William J. Rasdorf †

1 Introduction

System integration has become an important research issue in the interdisciplinary area of *computing in engineering*. System integration involves addressing the data communication and sharing needs of a set of application programs that automates related design activities. A common approach to system integration is the use of a central database management system for handling data communication and sharing needs [1, 2, 3, 4]. Relational databases have been used for this purpose. In a relational database data is represented in the tabular form. Each table represents a relationship among a set of attributes (objects). The relational view of data is particularly suitable for engineering applications, since engineering artifacts can be viewed in terms of objects and relations among objects. In this paper we describe an artifact modeling approach based on a relational view of the world that enables a more effective system integration for engineering applications from design and implementation standpoints.

To better deal with the complexities of system integration, it is imperative to address the design and implementation issues separately. In order to effectively design an integrated system we need to rely on a systematic approach that results in a general, robust, manageable, and flexible design. In this paper we identify such an approach based on the nature of the engineering design process and we illustrate the use of the approach for developing an integrated structural design system.

Engineering design can be viewed as a transformation process in which the initial description of an engineering system (artifact) transforms to its final description [5]. The automation of this process, therefore, requires 1) developing a computer-oriented model (herein called the artifact model) for describing the artifact and 2) formulating a set of operations that, when performed on the artifact model, transforms it from its initial (given) to its final (derived) form.

From the operational viewpoint, then, an automated system is composed of a set of application programs (transformation operators) such that each takes the artifact model as its input, modifies (transforms) the model, and gives the transformed model as its output. In this way, the artifact model transforms from its initial state to its final state. Herein, by initial state we mean a state in which only some of the attributes of the artifact have known values. By final state we mean the state in which all the attributes of the artifact have known values.

*Assistant Professor, Civil Engineering Department, Harbert Engineering Center, Auburn University, AL 36849

†Associate Professor, Civil Engineering Department, North Carolina State University, Raleigh, NC 27695

In order to develop an integrated system, then, we need to construct an artifact model that

- meets the data needs of and can be directly used by the application programs,
- correctly models the artifact,
- has the ability to evolve throughout the design process, and
- whose integrity can be automatically monitored and maintained.

This paper describes an artifact modeling approach that satisfies the above requirements and illustrates its use for developing an integrated system for the analysis, synthesis, and sizing stages of structural design. Our objective is to provide a uniform platform for supporting the engineering design process. We suggest that such a platform can effectively integrate the synthesis, analysis, and sizing stages of design as well as providing a common mechanism for different design disciplines to generate, represent, manipulate, and distribute design knowledge. Our focus, therefore, is on a modeling methodology to support automated engineering design.

1.1 The Modeling Approach

Engineering artifacts can be viewed as a set of entities (objects) and the relationships among them, where each object can be described in terms of its attributes. An engineering artifact can be modeled (represented) using a symbolic language that permits the representation of objects and relations, and is amenable to mechanical processing. The set of expressions of such a language that represents an artifact is the artifact model.

Our approach for constructing a consistent artifact model that helps integrate a set of application programs involves two major steps. The first step consists of identifying

- the set of artifact attributes relevant to each application program,
- the relationships between the attributes (the attributes and their relationships that are associated with an application program are an artifact submodel), and
- the relationships between the artifact submodels for maintaining the consistency and integrity of the whole model.

The second step, the focus of this paper, involves representing the attributes and their relationships using a symbolic language that is capable of maintaining the model's consistency and integrity.

1.2 Organization

The paper is organized in the following manner. In Section 2 a symbolic language for constructing artifact models is described. Section 3 addresses the representation of engineering artifact models using the language. In Section 4 the manipulation of artifact models is discussed. Section 5 provides a general discussion about the suggested approach.

2 A Language for Modeling Engineering Artifacts

In order to effectively construct artifact models a symbolic language that enables the relational representation and systematic manipulation of engineering knowledge is needed. One such language is an extended language of predicate logic. This section describes how predicate logic, in an extended form, can be used to represent, manipulate, and maintain the integrity of artifact models.

2.1 Predicate Logic

In order to capture knowledge using the language of predicate logic, knowledge has to be viewed as a collection of objects and relations. The language provides a set of constructs for representing primitive objects and relations and a set of syntax rules for formulating complex relations using the constructs. The language constructs, syntax rules, and an example of using them for knowledge representation in an engineering context are given below.

2.1.1 Constructs

Six types of constructs are used in the language of predicate logic. These are listed and described below.

1. **Object Symbols:** An object symbol is a construct that denotes an object or an object attribute. Lowercase letters are used to denote object symbols.
2. **Variable Symbols:** A variable symbol is a construct that denotes a variable over objects or object attributes. Variable symbols are shown using uppercase letters.
3. **Predicates:** A predicate is a construct that denotes a relation between a set of attributes. Each predicate has a predicate symbol followed by several terms enclosed in parentheses. For example, $equal(X, Y)$ is a predicate, which means $X = Y$. The attributes of a predicate are usually referred to as *terms*.
4. **Functions:** A function is a construct that denotes a mathematical function. Functions are similar to predicates in syntax, that is, a function is represented using a function symbol followed by a set of terms enclosed in parentheses. For example, $mult(2, X)$ is a function where $mult$ is the function symbol, and 2 and X are terms. In the classical predicate logic functions are uninterpreted constructs; they do not return any value.
5. **Logical Operators:** A logical operator is a construct that enables one to formulate complex relations (valid expressions of the language) using the other constructs. These are the commonly used logical operators: negation ($\neg$), disjunction ($\vee$), conjunction ($\wedge$), and implication ($\Rightarrow$).
6. **Quantifiers:** A quantifier is a construct that allows the inclusion of variables in expressions of the language. We use two quantifiers: the universal quantifier ($\forall$) and existential quantifier ($\exists$).

2.1.2 Syntax Rules

The valid expressions of the language are constructed using the following syntax rules.

1. A predicate is a valid expression if its terms are valid. A term is valid if it is a constant symbol, a variable symbol, or a function with valid terms.
2. If A is a valid expression, then $\neg A$ is a valid expression.
3. If A and B are valid expression, then $A \vee B$, $A \wedge B$, and $A \Leftarrow B$ are valid expression.
4. If X is a variable in A (a valid expression), then $\forall_X A$ and $\exists_X A$ are valid expressions.

2.1.3 Example

To demonstrate the use of predicate logic we draw upon a truss structure example. In particular, we will use a truss and represent its geometry, topology, and physical behavior using predicate logic. This example will be used throughout the paper.

Truss structure can be viewed as a collection of entities (objects) where each object can be described in terms of its attributes. Several object types are commonly used for modeling truss structures. These are truss element, load, support, and node object types.

Consider the fact that every truss element of a truss structure has to have two distinct end nodes. Only node object and truss element object types are needed for representing the above facts. Nodes can be represented using the following predicate:

$node(N, X, Y),$

where N is the node name, and X and Y are the coordinates of the node. Truss elements can be represented using the following predicate:

$element(I, N1, N2),$

where I is the truss element name, and $N1$ and $N2$ are names of the two end nodes of the element. Having defined these predicates, we can formulate the fact in this way:

$$\forall_I \forall_{N1} \forall_{N2} \forall_{X1} \forall_{Y1} \forall_{X2} \forall_{Y2} distinct\_nodes(I) \Leftarrow element(I, N1, N2) \wedge node(N1, X1, Y1) \wedge node(N2, X2, Y2) \wedge (X1 \neq X2 \vee Y1 \neq Y2)$$

The intuitive meaning of this expression is "truss element I has two distinct end nodes if the end nodes of I are $N1$ and $N2$, the coordinates of $N1$ are $X1$ and $Y1$, the coordinates of $N2$ are $X2$ and $Y2$, and either $X1 \neq X2$ or $Y1 \neq Y2$."

2.1.4 Extensions to Predicate Logic

To facilitate the representation and manipulation of design artifacts, we consider two extensions to the language of predicate logic. First, the language is extended such that arithmetic functions become a part of the vocabulary of the language. Thus, we have

$$plus(X, Y, X + Y)$$

as a valid expression of the extended language even though it is not a valid expression of the language of classical predicate logic (because of $X + Y$). This extension to the language of predicate logic, which was introduced by Jaffar and his co-workers [6], enables one to represent and manipulate arithmetic constraints. Therefore, it is suitable for engineering applications.

Second, the language is extended such that an object symbol or a collection of object symbols (called a *feature structure*) can be taken as a predicate term. This extension allows complex objects to be directly represented in predicates. The feature structure is a suitable means for representing and manipulating complex engineering objects. To better understand the utility of this extension to the language of predicate logic, consider the representation of a structural support object. In the classical language of predicate logic such an object can be represented using the following predicate:

$$support\_object(left\_support, pin, 0, 0),$$

where *left_support* is the object's name, *pin* is the support type, and "0" and "0" are the coordinates of the spatial position of the support. In the extended language this object can be represented using the following predicate:

$$object(X),$$

where X is a feature structure that takes the following form:

$$X \equiv \left[\begin{array}{ll} object\_type : & support \\ name : & left\_support \\ position : & \left[\begin{array}{l} x : 0 \\ y : 0 \end{array} \right] \\ support\_type : & pin \end{array} \right]$$

Note that the feature structure is a collection of attributes where each attribute has a name and a value. The value of an attribute can itself be a feature structure, as is the value of the predicate "*position*."

There are two main advantages of using feature structures for representing complex objects. First, the attributes themselves are referred to by name, rather than by position. For example, to refer to the type of the support object represented by the predicate *support_object*, one has to know that type appears as the second argument of the predicate. However, the position of the attribute in the feature structure is immaterial, the attribute is referred to by its symbolic name. Second, while the predicate *support_object* has a fixed arity (i.e., an arity of 4), the feature structure X is not fixed in size and can be expanded to include other attributes of interest. This is particularly suitable for design applications since in the early stages of problem solving process only a limited number of attributes exist. As the design process progresses, more attributes come into existence. In this manner then, the dynamic nature of the design process can be readily accommodated.

2.2 Knowledge Manipulation

Knowledge manipulation involves using an inference strategy to draw conclusions based on available knowledge. In the context of problem solving by drawing a conclusion we mean obtaining a valid solution to the problem at hand. In the previous section the use of predicate logic for knowledge representation was discussed. In this section we briefly discuss the use of a theorem proving strategy for knowledge manipulation.

The expressions of predicate logic that represent a problem of interest are called *axioms*. To solve the problem at hand, one needs to formulate an expression called a *theorem* that when proven true solves the problem. The truth or falsity of a theorem can be determined using an inference strategy known as resolution. To prove a theorem using this strategy, the theorem is first negated, then the negated theorem is proven to be false. Because the actual mechanics of the strategy is rather involved, it is not discussed here.

For a detailed discussion of the resolution strategy, see [7, 8, 9]. For a discussion of the resolution strategy for handling arithmetic expressions (the first extension mentioned in the previous section), see [6, 10]. For a discussion of the resolution strategy for handling feature structures (the second extension mentioned in the previous section), see [11]. A step-by-step example of the resolution strategy for an engineering application is given in [12].

3 Representing Artifact Models

The conceptual organization of an artifact model is best described in terms of an artifact data model, a set of artifact submodels, and the relationships between the submodels and the data model. The attributes of the artifact's objects are stored in an application-independent data structure. The data structure, with its contained data, is called the *artifact data model*. An artifact data model can be viewed as a database which contains the necessary data for representing a design artifact. The artifact attributes and their relationships associated with an application program comprise an artifact submodel. An artifact submodel can be viewed as the part of the database that contains the data needed by a particular application program. An artifact submodel represents the attributes of interest in a suitable manner for the use of the application program. The attribute values are not physically stored in the submodel; rather, they are deduced from the data model. The relationships between the submodels are established using a set of rules. These rules enable the data storage and retrieval from the artifact model. The conceptual view of an artifact model is shown in Figure 1.

Below are discussions of the use of predicate logic for formulating an artifact model in terms of a data model, examples of three submodels, and a specification of the relationships between the data model and the submodels.

3.1 The Artifact Data Model

An artifact data model is a general and application-independent representation of an artifact which is composed of objects. A feature structure is used for representing the attributes of an artifact

object. Thus, the artifact data model is the collection of feature structures representing the artifact's objects. Consider the truss structure shown in Figure 2. The structure is composed of two nodes, two supports, a truss element, and a load. At different stages of the design process different attributes of the objects become significant. For example, at the geometric modeling stage the geometry of the truss element and the location of the other objects are of interest. At the analysis stage the forces and displacements of the objects also become important. And, at the detailed design stages the allowable and induced stresses in the truss element become important. The feature structure is an expandable data structure that supports the representation of objects in all the design stages. As a result of the geometric modeling of the truss shown in Figure 2, the feature structures shown in Figure 3 come into existence. As a result of the analysis of the truss, the feature structures shown in Figure 3 evolve into the feature structures shown in Figure 4. And, as a result of the detailed design of the truss, the feature structure shown in Figure 4 evolves into the feature structure shown in Figure 5.

We define a unary predicate named "*fs*" for representing feature structures in the language of predicate logic. Using this predicate the artifact data model just discussed can be represented as shown below:

$fs(X_1)$
 $fs(X_2)$
 $fs(X_3)$
 $fs(X_4)$
 $fs(X_5)$
 $fs(X_6),$

where X_1 through X_6 are feature structures. Note that the data model evolves in the design process. That is, after the geometric modeling stage, X_1 through X_6 are the feature structures depicted in Figure 3; after the analysis stage, X_1 through X_6 are the feature structures depicted in Figure 4; and after the detailed-design stage, X_1 through X_6 are the feature structures depicted in Figure 5. The capacity of the model to support this evolution is critical.

The attribute of a feature structure can be referred to using built-in binary predicates of the form

$attribute\_name_{fs}(X, N),$

where $attribute\_name$ is the attribute's name, X is the feature structure, N is the attribute's value, and the subscript fs implies that the predicate's name is the name of a feature structure's attribute and indicates that the predicate is built-in. The predicate evaluates to true if N can be made syntactically identical to (unified with) the value of the attribute of interest; otherwise it evaluates to false. For example, consider the feature structure X_5 shown in Figure 3. The predicate $name_{fs}(X, N)$ returns true since the value of the attribute "name" can be unified with N (i.e., $e1 = N$). However, the predicate $name(X, e2)$ evaluates to false since $e2$ cannot be unified with $e1$.

3.2 Artifact Submodels

An artifact submodel has two functions: it makes the artifact attributes accessible to an application program, and it encapsulates the relationships among the artifact attributes so that the consistency of the attribute values and the integrity of the whole model can be maintained.

An application program views the artifact model as a set of objects, each with a fixed number of attributes. To accommodate the data needs of such a program, a set of predicates, each with a fixed number of attributes (terms), can be defined. Each occurrence of such a predicate represents an object, and its terms represent the object's attributes. This relational representation of data makes the data readily accessible to the application program and provides a framework for transferring data between the application program and the artifact model. Furthermore, the predicates enable a formal representation of the relationships between the attributes. In the remainder of this section the formulation of the three submodels for truss design systems are illustrated.

3.2.1 The Geometric Modeler Submodel

The first submodel for truss design systems is the geometric modeling submodel. In general, truss structures can be modeled using these four types of objects: nodes, supports, loads, and truss elements. Therefore, the geometry and topology of a truss structure can be described using the following four types of predicates:

```
node(Name, X, Y)
support(Name, Node, Type, Orient)
load(Name, Node, Force)
element(Name, Node1, Node2),
```

where

- the first predicate represents the name and coordinates of each node,
- the second predicate represents the name, node position, type, and spatial orientation of each support,
- the third predicate represents the name, position, spatial orientation, and magnitude of each load,
- and the last predicate represents the name and the two end nodes of each truss element.

For example, the geometry and topology of the truss structure shown in Figure 2 can be represented as shown below.

```
node(n1, 0, 0)
node(n2, 10, 0)
support(left, n1, pin, (0, 1))
```



```

support(right, n2, pin, (0, 1))
load(l, n2, [1, 0])
element(e1, n1, n2)

```

These four predicates allow the geometry, topology, applied loads, and boundary conditions of truss structures to be represented. The geometric modeling application program can easily interact with the artifact model using the four predicate types. Furthermore, the application-dependent relationships between the attributes of the predicates can be easily formulated and used to maintain the consistency and integrity of the model.

The geometric modeling of truss structures is an interactive process, thus subject to human error. To safeguard against such errors it is possible to define the application-dependent relationships that must exist between the components of truss structures, formulate these relationships using the language of predicate logic, and incorporate them into the artifact model, and therefore, into the corresponding submodel. These relationships, which are commonly referred to as integrity constraints, provide a means to verify the correctness of the modeled structure. For illustration purposes, several essential integrity constraints for geometric modeling of truss structures are presented below.

The first of these constraints, known as *constraintI*, is that every truss element has to have two distinct end nodes. This constraint is formulated in the following way:

$$\text{constraintI}(X) \leftarrow \text{element}(X, N1, N2) \wedge \text{node}(N1, X1, Y1) \wedge \text{node}(N2, X2, Y2) \wedge (X1 \neq X2) \vee (Y1 \neq Y2)$$

The intuitive interpretation of this expression is "*constraintI* is satisfied for every truss element *X* if 1) *N1* and *N2* are the two end nodes of the element; 2) *X1* and *Y1* are the coordinates of *N1*; 3) *X2* and *Y2* are the coordinates of *N2*; and 4) either *X1* $\neq$ *X2* or *Y1* $\neq$ *Y2*."

The second constraint, referred to as *constraintII*, is that every truss element must be connected to at least another truss or support element at each of its nodes. This constraint is formulated in the following manner:

$$\begin{aligned} \text{constraintII}(X) \leftarrow & \text{element}(X, N1, N2) \wedge \\ & (Y \neq X) \wedge (\text{element}(Y, N1, -) \vee \text{element}(Y, -, N1) \vee \text{support}(Y, N1, -, -)) \wedge \\ & (Z \neq X) \wedge (Z \neq Y) \wedge (\text{element}(Z, N2, -) \vee \text{element}(Z, -, N2) \vee \text{support}(Z, N2, -, -)) \end{aligned}$$

The intuitive interpretation of this expression is "*constraintII* is satisfied for truss element *X* if 1) *N1* and *N2* are the end nodes of *X*; 2) there is a truss element *Y* distinct from *X* with an end node *N1*; and 3) there is a truss element *Z* distinct from *X* and *Y* with an end node *N2*."

More complex integrity constraints can be easily defined using the language of predicate logic. For example, unstable truss structures can be identified using the following rules. Therefore, the existence of such structures in the data model can be maintained.

$$\neg \text{stable}(X) \leftarrow \text{concurrent\_reaction}(X) \vee \text{parallel\_reaction}(X) \vee \text{collapse\_mechanism}(X),$$

where X is the set of set names of the objects associated with the structure. This rule states that a structure is unstable if 1) its support reactions are either concurrent or parallel, or 2) if a collapse mechanism forms under a loading condition. This rule can be used to determine if the modeled truss structure is unstable. The rule is composed of these four predicates: "stable," "concurrent_reactions," "parallel_reactions," and "collapse_mechanism," where the first predicate is defined in terms of the other three. For illustration purposes we give the axiomatic definition of the predicate "concurrent." The other two predicates can be defined in a similar manner.

$$\begin{aligned} \text{concurrent\_reactions}(X) &\Leftarrow \text{support}(X, S) \wedge \text{concurrent}(S, Xc, Yc) \\ \text{concurrent}(X.T, Xc, Yc) &\Leftarrow \text{support}(X, N, \text{pin}, -) \wedge \text{node}(N, Xc, Yc) \wedge \text{concurrent}(T, Xc, Yc) \\ \text{concurrent}(X.T, Xc, Yc) &\Leftarrow \text{support}(X, N, \text{roller}, (I, J)) \wedge \text{node}(N, X0, Y0) \wedge \\ &((Yc - Y0) * I = (Xc - X0) * J) \wedge \text{concurrent}(T, Xc, Yc) \end{aligned}$$

The first rule states that the truss structure containing the set of objects X has concurrent reactions if S is the set of support objects in X and the lines of action of the support reactions are concurrent, (i.e., they all meet at a point with the coordinates Xc and Yc).

The second and third rules give the axiomatic definition of the predicate *concurrent* used in the first rule. The second rule verifies that the reactions are concurrent for a set of support objects where the first element of the set is a pin support. This is true if all such supports are located at the same point. The rule states that "for every object in the order set $X.T$, where X is the set's first element and T is the set of remaining elements, if X is a pin support located at node N , the coordinates of N are Xc and Yc , and the line of action of every element in T passes through the point $\langle Xc, Yc \rangle$, then all the reactions are concurrent."

The third rule verifies that the reactions are concurrent for a set of supports where the first element of the set is a roller. Note that the line of action of the roller reaction depends on the roller's inclination angle. Let $\langle I, J \rangle$ denote the orientation of the roller support such that when the vector is $\langle 0, 1 \rangle$, the reaction of the roller is vertical; when the vector is $\langle 1, 0 \rangle$, the reaction is horizontal; when the vector is $\langle 1, 1 \rangle$, the roller's reaction is inclined 45 degrees, and so on. Then, the equation of the line of action of the roller reaction can be expressed as

$$I * (Y - Y0) = J * (X - X0),$$

where $X0$ and $Y0$ are coordinates of the roller's position. Therefore, the third rule can be interpreted in the following manner. All the objects in the ordered set $X.T$, where X is a roller support, pass through the point $\langle Xc, Yc \rangle$ if 1) the location of X is node N , 2) the coordinates of N are $X0, Y0$, 3) the line of action of the roller's reaction passes through point $\langle Xc, Yc \rangle$, and 4) and the reaction of every element in T also passes through the same point.

3.2.2 The Analysis Submodel

The second submodel for truss design systems is the analysis submodel. The analysis application program generates new data base on the defined data in the geometric modeling stage. The new data includes the forces and displacements associated with the structural components. To represent

the new data in the artifact model, we use the following predicates, which are an extension of the predicates defined in the geometric submodel.

```
node(Name, X, Y)
support(Name, Node, Type, Orient, Force, Displacement)
load(Name, Node, Force, Displacement)
element(Name, Node1, Node2, Force, Displacement, E, A)
```

The added attributes to the predicates *support* and *load* are the force and displacement vectors associated with the support and load components, respectively. There are four attributes added to the predicate *element*. The first two represent the force and displacement vectors associated with the truss element, and the last two represent the element's modulus of elasticity and cross sectional area, respectively. These predicates can be used for representing the analyzed truss structures. For example, the truss structure in Figure 2, after being analyzed, can be represented as shown below.

```
node(n1, 0, 0)
node(n2, 10, 0)
support(left, n1, pin, (0, 1), [-1, 0], [0, 0])
support(right, n2, pin, (0, 1), [0, 0], [?, 0])
load(l, n2, [1, 0], [])
element(e1, n1, n2), [[1, 0], [-1, 0]], [[0, 0], [?, 0]], 29000, 4)
```

The non-interactive nature of the structural analysis process reduces the chance of generating inconsistent data due to human error. Nevertheless, the ability to verify the correctness of the generated data is highly desirable. The language of predicate logic enables the formulation of the set of rules that represents the relationships among the structural analysis data. These rules, which declaratively define the structure's equilibrium and compatibility equations, have two functions: they can be used to check the correctness of generated data, and they can be used for generating data. The formulation and implementation of these rules is respectively described in [13] and [14].

3.2.3 The Detailed Design Submodel

The third submodel for truss design systems is the detailed design submodel. This submodel can be defined in a way similar to the two previous submodels. The detailed design application program ensures that the designed truss complies with the appropriate design specifications. For the design of truss structures only one predicate needs to be defined in this submodel. This predicate represents the attributes of interest associated with truss elements, as shown below.

```
element(Name, Length, E, A, Rg, Fy, Force, Stress, Allowable_stress)
```

The relationships between these attributes can be easily represented using predicate logic. The relationships, then, can be used to verify the output of the detailed design application program or they can be used to actually perform the detailed design, as discussed in [15].

3.3 The Relationships Among Artifact Submodels

To complete the formulation of an artifact model, the relationships among the artifact data model and the artifact submodels need to be established. This section illustrates the use of predicate logic for representing such relationships.

The relationship between a data model and a submodel can be represented using rules, called transformation rules. These rules define the mapping between the artifact attributes in the submodel and the artifact attributes in the data model. For example, the following rule represents the relationship between the attributes of the predicate "node" in the geometric modeler submodel and the corresponding attributes in the data model.

$$\text{node}(N, X, Y) \Leftarrow fs(W) \wedge \text{type}_{fs}(W, \text{node}) \wedge \text{name}_{fs}(W, N) \wedge \text{position}_{fs}(W, P) \wedge \\ x_{fs}(P, X) \wedge y_{fs}(P, Y)$$

This rule can be intuitively interpreted in the following manner. A node object with name N and coordinates X and Y exists if these four conditions are met: 1) W is a feature structure with "type," "name," and "position" as its attributes, 2) the value of the attribute "type" is *node*, 3) the value of the attribute "name" is N , and 4) the value of the attribute "position" is a feature structure (denoted by P) with two attributes "x" and "y," where the value of the attribute "x" is X and the value of the attribute "y" is Y .

As another example consider the predicate "load" in the geometric modeler and analysis submodels. The relationship among the attributes of this predicate and the corresponding attributes in the data model can be established using the following two rules, one for each submodel.

$$\begin{aligned} \text{load}(\text{Name}, \text{Node}, [Fx, Fy]) &\Leftarrow fs(W) \wedge \text{type}_{fs}(W, \text{load}) \wedge \text{name}_{fs}(W, \text{Name}) \wedge \\ &\quad \text{position}_{fs}(W, \text{Node}) \wedge \text{force}_{fs}(W, P) \wedge x_{fs}(P, Fx) \wedge y_{fs}(P, Fy) \\ \text{load}(\text{Name}, \text{Node}, [Fx, Fy], [Dx, Dy]) &\Leftarrow fs(W) \wedge \text{type}_{fs}(W, \text{load}) \wedge \text{name}_{fs}(W, \text{Name}) \wedge \\ &\quad \text{position}_{fs}(W, \text{Node}) \wedge \text{force}_{fs}(W, P) \wedge x_{fs}(P, Fx) \wedge y_{fs}(P, Fy) \wedge \\ &\quad \text{displacement}_{fs}(W, D) \wedge x_{fs}(D, Dx) \wedge y_{fs}(D, Dy) \end{aligned}$$

What this illustrates is that the relationships between the data model and artifact submodels can be easily established using rules. These rules, which map the attributes of the submodels to those of the data model, can be used to store data in and retrieve data from the data model. The use of such rules for data storage and retrieval, and for maintaining the artifact model integrity is discussed in the following section.

4 Manipulating Artifact Models

Let us distinguish the conceptual description of the artifact model from its formal description. Conceptually the artifact model is composed of a data model, several submodels, and the relationships among the data model and the submodels as shown in Figure 1. Formally, the artifact model is a set of predicate logic expressions as shown in Section 3. As such, the artifact model is amenable to mechanical manipulation via the resolution strategy. An artifact model can be viewed

as a database management system capable of representing integrity constraints over its stored data. And, the resolution strategy can be viewed as the integrity maintenance mechanism of the database management system.

The logical expressions that represent an artifact model can be grouped into facts and rules. A fact is an expression with no logical operators, that is, a single predicate. A rule is an expression with several logical operators including one implication operator. The set of "*fs*" predicates which define the feature structures in the data model are facts. The relationships among submodel attributes (integrity constraints) and the relationships among the submodels and the data model are rules.

The manipulation of an artifact model manifests itself in formulating a theorem and proving it using the resolution strategy. In this section we first discuss two different modes of theorems and their interpretations. Then, we illustrate how data can be stored in and retrieved from the artifact model and how the consistency of the data can be checked and enforced by formulating and proving theorems.

4.1 Theorems and Their Interpretations

A theorem can be posed in two different modes; thus it has two interpretations. The first mode for posing a theorem is the checking mode. In this mode the theorem is proven true if it can be symbolically deduced from the artifact model (i.e., from the facts and rules). In this mode, the theorem can be interpreted as a question or query that is posed to the artifact model.

The second mode for posing a theorem is the enforcing mode. This mode is used when the lack of information prevents the theorem from being symbolically deduced from the artifact model. In the enforcing mode, the theorem is made true by adding the needed data to the artifact model. A theorem can be enforced if and only if it does not contradict the existing facts and rules in the artifact model. In this mode the theorem can be interpreted as a command to be executed. The symbol " $\uparrow$ " is used to indicate the enforcing mode of theorem proving. Let us illustrate the two modes of theorem proving using a simple example.

Suppose the artifact model is composed of one submodel which contains a single predicate named "*node*" with three attributes, "*Name*," "*X*," and "*Y*." The relationship among these attributes and the data model are defined using one transformation rule. Furthermore, suppose that the data model contains only one occurrence of the predicate "*fs*," as shown in Figure 6. This artifact model represents a single node.

Consider the following theorem which is specified in the checking mode.

$node(n1, 0, 0)$

The intuitive interpretation of this theorem is the query "Does the artifact model contain the representation of a node named *n1* with the coordinates 0 and 0?" The answer to this question is *yes*, since the theorem can be shown to be a logical consequence of the artifact model.

Now, consider the following theorem which is specified in the enforcing mode.

$\uparrow node(n2, 10, 0)$

The intuitive interpretation of this theorem is "If not contradictory with the artifact model, there must exist a node with name $n2$ and coordinates 10 and 0." Although the theorem is not a logical consequence of the artifact model, it can be made so by adding additional data to the artifact model, since the theorem does not contradict the model. As a result of enforcing this theorem the artifact model evolves, as depicted in Figure 7.

Finally, consider the theorem

$\uparrow \text{node}(n1, 0, 10)$

This theorem cannot be enforced since it contradicts the present state of the artifact model. That is, since the node $n1$ has the coordinates (0,0), it cannot take the coordinates (0,10).

The checking mode of theorem proving can be used to retrieve data from and to check the integrity constraints associated with the artifact model. The enforcing mode of theorem proving can be used to store data in and enforce the integrity constraints associated with the artifact model.

4.2 Data Storage

A request for data storage is issued by an application program. The artifact model receives the request from the application program in the form of a theorem. The theorem can be enforced; thus the data can be added to the model if the new data does not contradict the existing data. For example, the geometric modeler application program, through its interface, can pose the following theorems.

$\uparrow \text{node}(n1, 0, 0)$
 $\uparrow \text{node}(n2, 10, 0)$
 $\uparrow \text{load}(l2, n2, [1, 0])$
 $\uparrow \text{support}(\text{left}, n1, \text{pin}, (1, 0))$
 $\uparrow \text{support}(\text{right}, n2, \text{roller}, (1, 0))$
 $\uparrow \text{element}(e1, n1, n2)$

As a result of proving these theorems the data specified in the theorem is added to the artifact model. The added data represents the truss structure shown in Figure 2.

4.3 Data Retrieval

A request for data retrieval is issued by an application program and passed to the artifact model via an interface program. The request takes the form of a checking theorem in the artifact model. If the theorem can be proven true, then the retrieved data is returned to the application program; otherwise a message indicating that the requested data does not exist is returned. For example, the analysis application program may send the following data request to the artifact model:

$\text{element}(n1, L, E, A, F, D)$

Based on the available data in the data model, all or some of the attributes instantiate to known values, and the retrieved data is returned to the application program. According to our formulation, only L (length) of the element can be instantiated to a known value. The remaining attributes are returned to the application program as variables. This does not mean that the other attributes cannot be deduced from the model; rather, it means that the values of the other attributes are not yet available in the artifact model.

4.4 Constraint Checking

Constraint checking can be accomplished by formulating theorems in the checking mode. Such theorems are composed of a set of predicates that are separated by logical operators. For example, after the geometric modeler application program data is stored in the artifact model, the consistency of the data with respect to the two constraints, *constraintI* and *constraintII*, can be checked using the following theorem.

$$\text{constraintI}(e1) \wedge \text{constraintII}(e1)$$

This theorem, if proven true, implies that the end nodes of the truss element $e1$ are distinct and the truss element is connected to at least one other element at each of its end nodes.

4.5 Constraint Enforcement

Constraint enforcement can be accomplished by formulating theorems in the enforcement mode. Such a formulated theorem can be made true by automatically generating and adding the necessary data to the artifact model. For example, constraint enforcement can be used to generate the forces and displacements in truss members (i.e., to analyze trusses) by manipulating the artifact model without invoking the analysis application program.

For the purpose of illustration let the predicate *analysis(X.T)* stand for the set of integrity constraints associated with the analysis submodel of a truss design system, such that when the relationships between the submodel attributes are satisfied, the predicate evaluates to true. Then, the theorem

$$\uparrow \text{analysis}(X.T),$$

where $X.T$ is the set of truss objects, can be posed to generate the forces and displacements of the truss objects. That is, this theorem can be used to analyze the truss structure.

5 Discussion

This paper presents a simple yet powerful approach for representing engineering artifacts. This approach involves using a symbolic language that enables engineering objects and relations to be represented and that is amenable to mechanical processing. The approach provides a common

platform for representing diverse engineering knowledge, and thus supports and facilitates system integration.

The artifact model of an engineering system provides an application-independent representation of the system in the sense that the organization of data in the model is independent of the organization of the same data in the application programs. Still, it accommodates the data needs of its associated application programs and maintains the consistency and integrity of its contained data.

An artifact model interacts with application programs through a set of program interfaces. Requests for data storage and retrieval are sent to the artifact model via these interfaces. Figure 8 shows the organization of an integrated system composed of an artifact model and a set of application programs.

A new application program can be added to the system by 1) formulating a submodel that represents the objects and attributes used by the application program and 2) developing an interface between the artifact model and the application program such that data can be transferred between the two.

In general an artifact model can be interfaced with any program or system. For example, the described artifact model can be interfaced with a relational database management system for structural design, thus becoming a part of it. The artifact model adds the needed ability to define and maintain integrity constraints on the data in the database.

Furthermore, viewed as an stand-alone system, an artifact model is a knowledge-based system for solving the problem at hand. This is so because the system captures the domain knowledge, in the form of integrity constraints and feature structures, and has the ability to manipulate knowledge for solving the problem.

An artifact model developed using predicate logic is a versatile tool for solving an engineering problem. It can be attached to a set of application programs to facilitate their integration, and it can be attached to relational database management systems for enhancing their constraint maintenance abilities.

5.1 Future Work

The approach presented here can be expanded in several directions. First, to illustrate the utility and applicability of the presented methodology for integrating complex engineering systems a programming environment that supports the construction of artifact models and facilitates the interaction of the model with external programs is needed. We believe that such an environment can be developed by complementing an existing constraint logic programming language such as CLP(R) with the needed functionalities, such as feature structures.

Second, an extensive effort to identify the constraints associated with design processes is needed. Substantive research in this area is surely lacking. By identifying and incorporating such design knowledge in the artifact model, one makes the model knowledgeable about the domain of interest. Thus, pertinent decisions can be made intelligently, yet mechanically, by the model itself. And, a degree of automation not previously obtained will be achieved.

Third, to fully explore the applicability, and to identify shortcomings, of the presented approach

several real-size case studies need to be conducted. Such case studies can also be used to obtain scientific statistical data on the time and space efficiency of the approach.

5.2 Acknowledgment

This work was supported, in part, by the National Science Foundation under MSM-8451465. The support of NSF is gratefully acknowledged.

References

- [1] C.A. Fellipa. Database management in scientific computing – i. general description. *Computers and Structures*, 10:53–61, 1979.
- [2] A.R. Kutay. *Multi-User Concurrent Design Databases*. PhD thesis, Institute of Building Sciences Report No. 20, Carnegie Mellon University, August 1981.
- [3] P.A. Fishwick and C.L. Blackburn. The integration of engineering programs using a relational database scheme. In *Second International Computer Engineering Conference*, pages 173–181, 1982.
- [4] Y.E. Kalay. A database management approach to cad/cam system integration. In *22nd ACM/IEEE Design Automation Conference*, pages 111–116, 1985.
- [5] V. Hubka and W. E. Eder. A scientific approach to engineering design. *Design Studies*, 8(3):123–137, 1987.
- [6] J. Jaffar and S. Michaylov. Methodology and implementation of a clp system. In *Fourth International Conference on Logic Programming*, pages 196–218, 1986.
- [7] R.A. Robinson. A machine-oriented logic based on resolution principle. *Journal of Association for Computing Machinery*, 12(1):23–41, 1965.
- [8] D. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, 1978.
- [9] N.R. Genesereth and N.J. Nilsson. *Logical Foundation of Artificial Intelligence*. Morgan Kaufmann, 1988.
- [10] J. Jaffar and J-L. Lassez. From unification to constraints. In *Sixth International Conference on Logic Programming*, pages 1–18, 1987.
- [11] K. Knight. Unification: A multidisciplinary survey. *ACM Computing Survey*, 21(1):93–124, 1989.
- [12] S. Lakmazaheri. *A study on the Constraint Logic Approach for Structural Design Automation*. PhD thesis, North Carolina State University, Raleigh, North Carolina, August 1990.
- [13] S. Lakmazaheri and W.J. Rasdorf. The analysis and partial synthesis of truss structures via theorem proving. *Engineering with Computers*, 6(1):31–45, 1990.

- [14] S. Lakmazaheri and W.J. Rasdorf. Constraint logic programming for the analysis and partial synthesis of truss structures. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 3(3):157-173, 1989.
- [15] W.J. Rasdorf and S. Lakmazaheri. A logic-based approach for processing design standards. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 4(3), 1990.

Figure 1: The Conceptual View of an Artifact Model

Figure 2: A Simple Truss Structure

$$\begin{aligned}
X_1 &\equiv \begin{bmatrix} \text{type :} & \text{node} \\ \text{name :} & n1 \\ \text{coordinates :} & \begin{bmatrix} x : 0 \\ y : 0 \end{bmatrix} \end{bmatrix} & X_2 &\equiv \begin{bmatrix} \text{type :} & \text{node} \\ \text{name :} & n2 \\ \text{coordinates :} & \begin{bmatrix} x : 120 \\ y : 0 \end{bmatrix} \end{bmatrix} \\
X_3 &\equiv \begin{bmatrix} \text{type :} & \text{support} \\ \text{name :} & \text{left} \\ \text{position} & n1 \\ \text{support\_type :} & \text{pin} \end{bmatrix} & X_4 &\equiv \begin{bmatrix} \text{type :} & \text{support} \\ \text{name :} & \text{right} \\ \text{position} & n2 \\ \text{support\_type :} & \text{roller} \end{bmatrix} \\
X_5 &\equiv \begin{bmatrix} \text{type :} & \text{truss\_element} \\ \text{name :} & e1 \\ \text{position :} & \begin{bmatrix} \text{node1 : } n1 \\ \text{node2 : } n2 \end{bmatrix} \end{bmatrix} & X_6 &\equiv \begin{bmatrix} \text{type :} & \text{load} \\ \text{name :} & l1 \\ \text{position} & n2 \\ \text{force :} & \begin{bmatrix} x : 1 \\ y : 0 \end{bmatrix} \end{bmatrix}
\end{aligned}$$

Figure 3: The Artifact Data Model After the Geometric Modeling Stage

$$\begin{aligned}
X_1 &\equiv \begin{bmatrix} \text{type :} & \text{node} \\ \text{name :} & n1 \\ \text{coordinates :} & \begin{bmatrix} x : 0 \\ y : 0 \end{bmatrix} \end{bmatrix} & X_2 &\equiv \begin{bmatrix} \text{type :} & \text{node} \\ \text{name :} & n2 \\ \text{coordinates :} & \begin{bmatrix} x : 120 \\ y : 0 \end{bmatrix} \end{bmatrix} \\
X_3 &\equiv \begin{bmatrix} \text{type :} & \text{support} \\ \text{name :} & \text{left} \\ \text{position :} & n1 \\ \text{support\_type :} & \text{pin} \\ \text{displacement :} & \begin{bmatrix} x : 0 \\ y : 0 \end{bmatrix} \\ \text{force :} & \begin{bmatrix} x : -1 \\ y : 0 \end{bmatrix} \end{bmatrix} & X_4 &\equiv \begin{bmatrix} \text{type :} & \text{support} \\ \text{name :} & \text{right} \\ \text{position :} & n2 \\ \text{support\_type :} & \text{roller} \\ \text{displacement :} & \begin{bmatrix} x : 0.001 \\ y : 0 \end{bmatrix} \\ \text{force :} & \begin{bmatrix} x : 0 \\ y : 0 \end{bmatrix} \end{bmatrix} \\
X_5 &\equiv \begin{bmatrix} \text{type :} & \text{truss\_element} \\ \text{name :} & e1 \\ \text{support\_type :} & \text{roller} \\ \text{position :} & \begin{bmatrix} \text{node1 :} n1 \\ \text{node2 :} n2 \end{bmatrix} \\ \text{area :} & 4 \\ \text{modulus :} & 29000 \\ \text{displacement :} & \begin{bmatrix} \text{node1 :} \begin{bmatrix} x : 0 \\ y : 0 \end{bmatrix} \\ \text{node2 :} \begin{bmatrix} x : 0.001 \\ y : 0 \end{bmatrix} \end{bmatrix} \\ \text{force :} & \begin{bmatrix} \text{node1 :} \begin{bmatrix} x : -1 \\ y : 0 \end{bmatrix} \\ \text{node2 :} \begin{bmatrix} x : 1 \\ y : 0 \end{bmatrix} \end{bmatrix} \end{bmatrix} \\
X_6 &\equiv \begin{bmatrix} \text{type :} & \text{load} \\ \text{name :} & l1 \\ \text{position :} & n2 \\ \text{force :} & \begin{bmatrix} x : 1 \\ y : 0 \end{bmatrix} \\ \text{displacement :} & \begin{bmatrix} x : 0.001 \\ y : 0 \end{bmatrix} \end{bmatrix}
\end{aligned}$$

Figure 4: The Artifact Data Model After the Analysis Stage

$$\begin{aligned}
X_1 &\equiv \begin{bmatrix} \text{type :} & \text{node} \\ \text{name :} & \text{n1} \\ \text{coordinates :} & \begin{bmatrix} x : 0 \\ y : 0 \end{bmatrix} \end{bmatrix} & X_2 &\equiv \begin{bmatrix} \text{type :} & \text{node} \\ \text{name :} & \text{n2} \\ \text{coordinates :} & \begin{bmatrix} x : 120 \\ y : 0 \end{bmatrix} \end{bmatrix} \\
X_3 &\equiv \begin{bmatrix} \text{type :} & \text{support} \\ \text{name :} & \text{left} \\ \text{position :} & \text{n1} \\ \text{support\_type :} & \text{pin} \\ \text{displacement :} & \begin{bmatrix} x : 0 \\ y : 0 \end{bmatrix} \\ \text{force :} & \begin{bmatrix} x : -1 \\ y : 0 \end{bmatrix} \end{bmatrix} & X_4 &\equiv \begin{bmatrix} \text{type :} & \text{support} \\ \text{name :} & \text{right} \\ \text{support\_type :} & \text{roller} \\ \text{position :} & \text{n2} \\ \text{displacement :} & \begin{bmatrix} x : 0.001 \\ y : 0 \end{bmatrix} \\ \text{force :} & \begin{bmatrix} x : 0 \\ y : 0 \end{bmatrix} \end{bmatrix} \\
X_5 &\equiv \begin{bmatrix} \text{type :} & \text{truss\_element} \\ \text{name :} & \text{e1} \\ \text{position :} & \begin{bmatrix} \text{node1 : n1} \\ \text{node2 : n2} \end{bmatrix} \\ \text{area :} & 4 \\ \text{modulus :} & 29000 \\ \text{displacement :} & \begin{bmatrix} \text{node1 :} \begin{bmatrix} x : 0 \\ y : 0 \end{bmatrix} \\ \text{node2 :} \begin{bmatrix} x : 0.001 \\ y : 0 \end{bmatrix} \end{bmatrix} \\ \text{force :} & \begin{bmatrix} \text{node1 :} \begin{bmatrix} x : -1 \\ y : 0 \end{bmatrix} \\ \text{node2 :} \begin{bmatrix} x : 1 \\ y : 0 \end{bmatrix} \end{bmatrix} \\ \text{stress :} & 0.25 \\ \text{allowable\_stress :} & 14.4? \end{bmatrix} \\
X_6 &\equiv \begin{bmatrix} \text{type :} & \text{load} \\ \text{name :} & \text{l1} \\ \text{position :} & \text{n2} \\ \text{force :} & \begin{bmatrix} x : 1 \\ y : 0 \end{bmatrix} \\ \text{displacement :} & \begin{bmatrix} x : 0.001 \\ y : 0 \end{bmatrix} \end{bmatrix}
\end{aligned}$$

Figure 5: The Artifact Data Model After the Detailed Design Stage

$$fs\left(\begin{array}{ll} type : & node \\ name : & n1 \\ position : & \begin{bmatrix} x : 0 \\ y : 0 \end{bmatrix} \end{array}\right)$$

$$node(N, X, Y) \Leftarrow fs(W) \wedge type_{fs}(W, node) \wedge name_{fs}(W, N) \wedge position_{fs}(W, P) \wedge \\ x_{fs}(P, X) \wedge y_{fs}(P, Y)$$

Figure 6: An Artifact Model

$$\begin{array}{l}
fs\left(\begin{array}{l} type : \quad node \\ name : \quad n1 \\ position : \quad \left[\begin{array}{l} x : 0 \\ y : 0 \end{array} \right] \end{array} \right) \\
fs\left(\begin{array}{l} type : \quad node \\ name : \quad n2 \\ position : \quad \left[\begin{array}{l} x : 10 \\ y : 0 \end{array} \right] \end{array} \right)
\end{array}$$

$$node(N, X, Y) \Leftarrow fs(W) \wedge type_{fs}(W, node) \wedge name_{fs}(W, N) \wedge position_{fs}(W, P) \wedge x_{fs}(P, X) \wedge y_{fs}(P, Y)$$

Figure 7: The Modified Artifact Model

Figure 8: The Organization of an Integrated System

Contents

1	Introduction	2
1.1	The Modeling Approach	3
1.2	Organization	3
2	A Language for Modeling Engineering Artifacts	4
2.1	Predicate Logic	4
2.1.1	Constructs	4
2.1.2	Syntax Rules	5
2.1.3	Example	5
2.1.4	Extensions to Predicate Logic	6
2.2	Knowledge Manipulation	7
3	Representing Artifact Models	7
3.1	The Artifact Data Model	7
3.2	Artifact Submodels	9
3.2.1	The Geometric Modeler Submodel	9
3.2.2	The Analysis Submodel	11
3.2.3	The Detailed Design Submodel	12
3.3	The Relationships Among Artifact Submodels	13
4	Manipulating Artifact Models	13
4.1	Theorems and Their Interpretations	14
4.2	Data Storage	15
4.3	Data Retrieval	15
4.4	Constraint Checking	16
4.5	Constraint Enforcement	16
5	Discussion	16
5.1	Future Work	17
5.2	Acknowledgment	18