Proceedings of DETC'03
2003 ASME Design Engineering Technical Conferences &
Computers and Information in Engineering Conference
September 2–6, 2003, Chicago, Illinois

DETC03/CIE-48233

EXTENSIBLE SEMANTICS FOR REPRESENTING ELECTROMECHANICAL ASSEMBLIES

Joseph Kopena **William C. Regli**
Geometric and Intelligent Computing Laboratory
Computer Science Department, Drexel University
3141 Chestnut Street
Philadelphia, PA, 19104

ABSTRACT

Many representations exist for describing engineering artifacts. These representations come from a wide variety of work in many research areas with differing backgrounds and purposes. No unified representation has emerged combining aspects of existing schemes with an ability to easily grow and be extended over time. In this work we demonstrate the application of methods from the knowledge representation community to the domain of engineering artifacts, in particular electromechanical assemblies. We use conceptual graphs as a formal, cleanly extensible representation form. Aspects of several existing results are combined to create a unified representation and demonstrate the approach. We also discuss translation of the conceptual graphs to the DARPA Agent Markup Language (DAML) in order to utilize and cooperate with efforts to develop ontologies for this domain.

Introduction

This research describes a technique to adapt and extend the formal models from the knowledge representation community to develop a semantically grounded methodology for representing electromechanical assembly models. This is a highly multi-disciplinary problem, where previous research has integrated ideas from artificial intelligence with concepts from computational geometry, solid modeling, robotics and engineering design. The common approach is to take the mathematical models of the geometric world and act upon them with AI techniques: operating on assembly features, geometry/topology, and constraints with rules, planners, search, and knowledge-bases to perform activities such as path planning, process planning, assembly sequencing, and design analysis. Viewed collectively, each of these research results is an isolated effort.

This paper shows how the present generation of knowledge representation techniques can be used to achieve new levels of semantic interoperablity. The lack of interoperablity stems from the the lack of a unified representation scheme capable of incorporating elements of representations from many backgrounds and applications as well as being extensible to meet future demands. Standardization efforts such as ISO STEP have achieved a degree of *shared syntax* through which such lower-level design data elements as boundary representation models can be shared across CAD systems. However, in this exchange the semantics of the design models are lost and one is left with only the shadow of the original artifact. Further, even when inside the original CAD system, semantics are represented as a patchwork of mathematics, constraints and features—not as expressions grounded in a shared ontology with known semantics.

Our approach is twofold: First, to use *conceptual graphs* to create a foundation in which we can connect a wide variety of other electromechanical assembly representations. Second, to use the emerging languages of the *Semantic Web*, in particular the DARPA Agent Markup Language (DAML), as the means of implementing our approach and executing experiments, example queries and operations on these graphs. Using DAML allows us to take advantage of its many other ontology and tool development efforts, enabling our representations to be extended by and

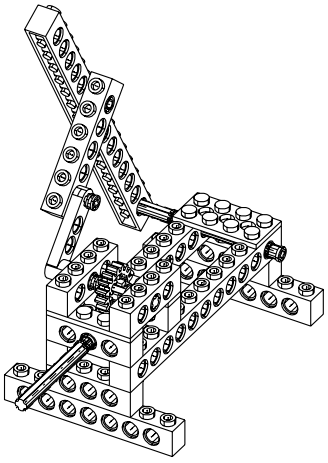


Figure 1. Example LEGO mechanical assembly—a windshield wiper.

used by others in the wider DAML community.

Throughout the paper, we use examples from the domain of electromechanical devices created with LEGO^{TM} Technic components, such as illustrated in Figure 1. The LEGO domain represents a restricted, yet complex, domain of "real" engineering artifacts: the quantity and diversity of component classes provides for construction of a wide and complex range of mechanisms suitable for our purposes.

The major contribution of this paper is to broaden and deepen our abilities to interrogate engineering design knowledge. The conceptual graph formalization, implemented in DAML, enables new types of translators, knowledge brokers and integrators to be developed. Specifically, the paper shows how we can now perform queries on design information at multiple levels of abstraction and across representation domain (e.g., queries that simultaneously ask about function and assembly structure). With the existing heterogeneous mix of CAD/CAE environments, we see this work as one step toward developing a shared semantics through which engineering knowledge can be objectively represented. Our ultimate goal is to truly broaden the term Geometric Reasoning to encompass knowledge that extends beyond the manipulation of mathematics to include function, behavior, relationships and design rationale.

This paper is structured as follows: first, we overview some related work on assembly representation. Then, we describe our approach to assembly representation based on conceptual graphs and show how this can be the basis for the development of a more detailed assembly ontology. Next, we give several detailed examples of the representation in use and describe operations we can apply to it. Lastly, we discuss our use of DAML and provide a summary of our contributions and conclusions.

Related Work

This section briefly overviews four areas of assembly representation, roughly drawn from engineering design, artificial intelligence, and computer-aided design. Each of these areas has focused on representing different, often orthogonal, aspects of assemblies.

Representing Mechanical Function. Several conclusions have been put forth by recent work representing assemblies and their function (Kim, 1997; Bose et al., 1997; Umeda and Tomiyama, 1997)): (1) CAD is still primarilly geometric; (2) most CAD systems do not bother to represent or capture function. In an effort to address these problems, function and flow representations (Szykman et al., 1999a; Hirtz et al., 2001; Kirschman et al., 1996) work explicitly on the notion of capturing the functions present in an assembly. These techniques for modeling function have been greatly influenced by the case-based reasoning and case-based design communities (Fowler, 1996).

Logic-based Descriptions and Qualitative Physics. Much of the core AI research in this category comes from the qualitative physics community (Faltings, 1990; Kuipers, 1984; Forbus, 1984) and declarative representations such as (Subramanian and Wang, 1993). The goal of each is to represent objects and associated physical reasoning (i.e., gravity, friction, motion, etc.). As such, they provide a semantically pleasing description of the forces of the world. However, little has been done to rigorously integrate these representations with detailed CAD or engineering-oriented models of behavior.

Composable Simulation Models. Composable port based modeling (Sinha et al., 2001) attempts to provide for modeling the behavior of devices at varying levels of detail. Each component and interaction in a model is associated with a set of mathematical models describing its behavior at varying levels of abstraction. Interactions occur at ports, finite patches of geometry on solid models associated with the components. This system allows the designer to choose what level of simulation to perform to suit particular needs and to provide support throughout the design process by allowing the models to be gradually enriched as the design is refined.

CAD and Constraint-Based Modeling. Most engineers are familiar with representing assemblies in a computer-aided design package. At the core of these representations are the solid models of the assembly components. Each CAD package has a particular set of component interactions which it is designed to handle. Some of these are assembly relations such as fits, mate, and align, all of which describe a rigid body. These conditions can be described as relationships on the geometric shapes which make up the components. Other interactions are joints and movements, such as revolute joints and translations which describe a mechanical assembly. These mechanics can be specified as a set of analytical geometry equations. By solving and analyzing these relations and equations the components can be positioned

exactly and the mechanism simulated (Lee and Andrews, 1985; Hoffmann and Joan-Arinyo, 1997; Anantha et al., 1996).

Technical Approach

A *conceptual graph* is defined as a finite, directed, connected, bipartite, labeled multigraph $G = (V, E, l)$ and is described extensively in (Sowa, 1984) and (Sowa, 2001). The two vertex sets of the graph consist of concept nodes, C, and conceptual relation nodes, R, both labeled by l. The term "concepts" is derived from the field's roots in natural language processing and cognitive science. These nodes are abstract objects between which relations can be made and we will refer to them in that fashion in subsequent sections of this paper. Conceptual relation nodes make connections between concepts, and no two concepts or two conceptual relations are directly connected.

The labels of the conceptual relation nodes come from the terminology $\mathbf{R}$ of defined labels to which an agent can commit. Concept node labels include the type of the concept as well as quantifier information. Existential quantification is assumed in the absence of specific instance identification. The type of a concept is drawn from a terminology $\mathbf{C}$ of types:

$$\forall v \in C \quad \text{type}(v) \in \mathbf{C}$$

A hierarchy representing subclass relationships can be created by partially ordering the terminology of types with a statement of the form $\mathbf{C_i} \subset \mathbf{C_j}$ such that the following holds:

$$\forall v \in C, \mathbf{C_i} \in \mathbf{C}, \mathbf{C_j} \in \mathbf{C} \quad \text{type}(v) = \mathbf{C_i} \wedge \mathbf{C_i} \subset \mathbf{C_j} \Rightarrow \text{type}(v) = \mathbf{C_j}$$

A concept may be an instance of several types as result of such sub and super-class relationships or as a result of multiple inheritance. Types can also be defined using a conceptual graph representing necessary conditions for membership in the type. These terminologies along with the set of type definitions form the *support* of the conceptual graph—the domain ontology.

Example. A commonly used piece of information associated with engineering assembly is that of component adjacencies. This is represented by a *contact graph* $G = (V, E)$ where V is a set of components in an assembly and each edge in E represents an intersection or mating contact between components. More formally a contact graph is defined by the following:

$$\forall u \in V, v \in V \quad (u, v) \in E \Leftrightarrow \text{pointset}(u) \cap \text{pointset}(v) \neq \emptyset$$

This contact graph has a clear translation into a conceptual graph. The two graphs share a common vertex set, objects representing components in the assembly. Each edge in the contact graph represents a particular relation in the conceptual graph, *contact*. We can commit to a meaning of this relation such that the conceptual graph equates to the contact graph:

$$\forall u \in \mathbf{C}, v \in \mathbf{C}, r \in \mathbf{R}$$
$$(u, r) \in E \wedge (r, v) \in E \wedge \text{label}(r) = \text{"contact"} \Leftrightarrow \text{pointset}(u) \cap \text{pointset}(v) = \emptyset$$

Such a conceptual graph relating the contact graph for the example LEGO assembly of Figure 1 is shown in Figure 2(a). This conceptual graph can be easily extended to incorporate new information in addition to the contact relationships. For example, a new type can be incorporated into $\mathbf{C}$ for labeling objects to represent an abstract notion of assemblies in addition to components. A new relation label *part* can also be incorporated into $\mathbf{R}$ for relating an assembly to its components and subassemblies. With these terms we can incorporate more information into the conceptual graphs depicting contact relations. Figure 2(b) shows the contact graph of our example assembly extended to include a hierarchy of subassembly groupings.

Representing Assemblies

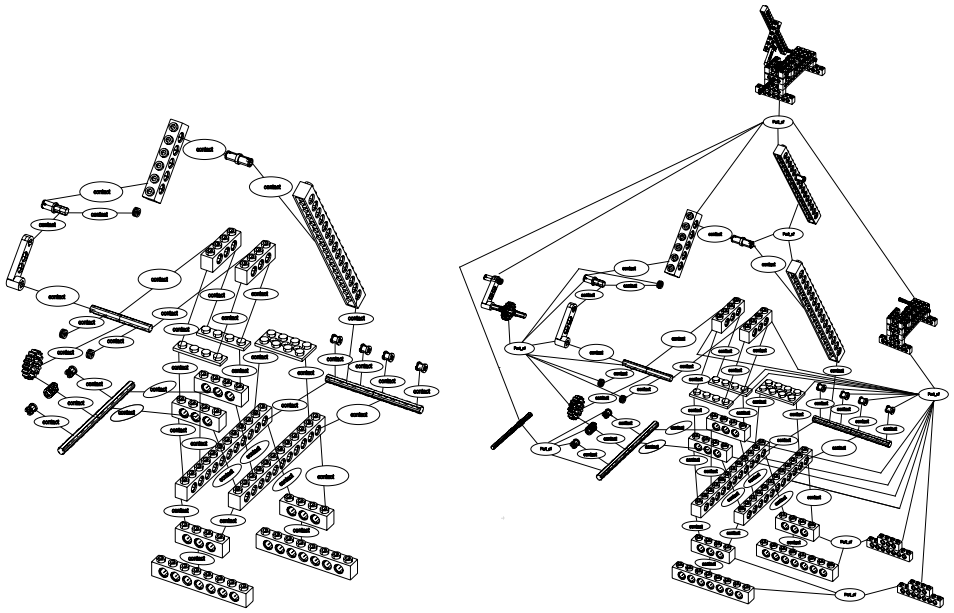
This section develops a more extensive ontology of relations and object definitions for representing engineering assemblies with conceptual graphs. This ontology includes aspects of several distinct representation forms including CAD and detailed physical modeling, composable port-based modeling, function and flow, and information specific to the components and assemblies making up our universe of Lego parts. Within the conceptual graph framework all of these elements can be unified and easily extended in the future to include other information.

Representing Components

To represent an assembly in any detail it is necessary to describe its components, objects of the *Component* class. In particular, it will be useful to model the features of the component through which it interacts with other elements in the assembly. For our domain of LEGO pieces the primary features are assembly features, the holes, shafts, and planar faces which control how the pieces fit together. A typical piece is depicted in Figure 3.

These features can be included as object classes in the type hierarchy of our conceptual graphs, $\mathbf{C}$. They can be further defined through relating them to their geometric shapes. Through use of the subset relation $\subset$ between types we can build a hierarchy of features and their shapes as in Figure 4.

Further information can be applied to the description through the use of type definitions containing conditions necessary for membership to the type. Since we are largely interested in the shape of the features, we define properties of the



(a) Contact graph as conceptual graph. (b) Contact graph and subassembly hierarchy.

Figure 2. Conceptual graph representation of example assembly in Figure 1.

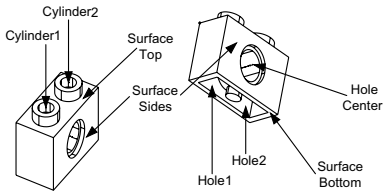


Figure 3. Assembly features of a typical brick.

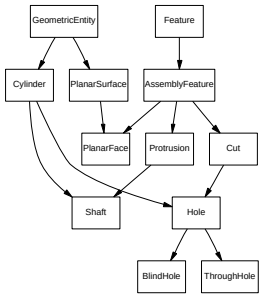


Figure 4. Relationships between assembly features and their shape.

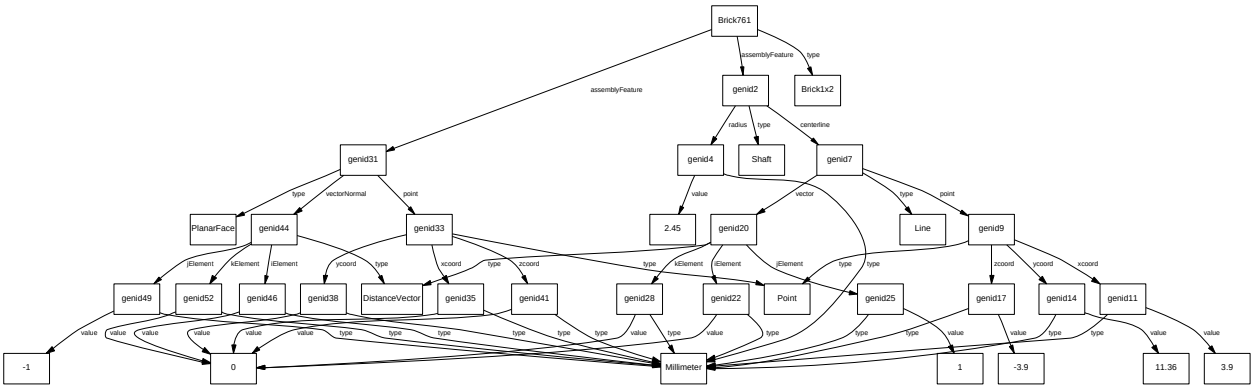
associated geometric entities. The following conceptual graphs define properties for some of these classes:

```
type Cylinder(x) is
    [GeometricEntity:*x] –
        → (centerLine) → [Line]
        → (radius) → [Distance].
type Line(x) is
    [GeometricEntity:*x] –
        → (point) → [Point]
        → (vector) → [DistanceVector].
type Point(x) is
    [GeometricEntity:*x] –
        → (xcoord) → [Distance]
        → (ycoord) → [Distance]
        → (zcoord) → [Distance].
```

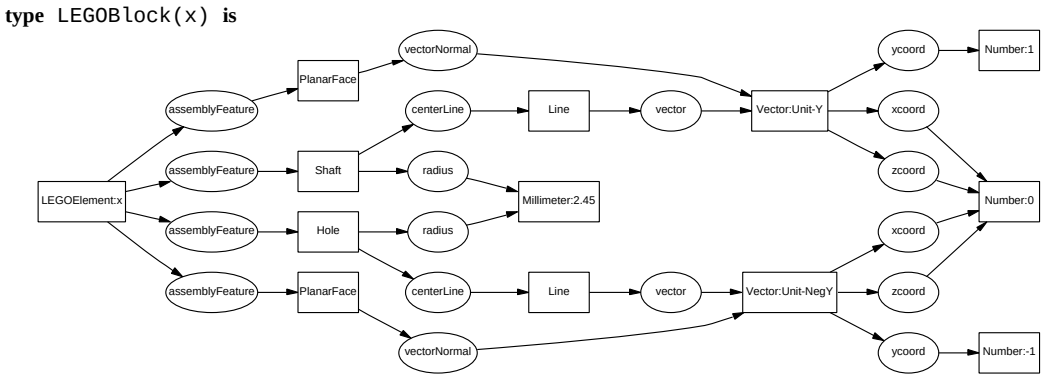
Following these definitions we can construct conceptual graphs depicting the features of our components. Figure 5(a) depicts such a graph including some of the features of the example brick shown in Figure 3. Note that in this modeling all values have been typed, in this case as millimeter units.

It is important to realize that the number of classes and information included in this representation of features can be easily extended. The hierarchy of features and shapes can grow through adding new terms to **C** and relating them to existing items. New properties and definitions can be introduced to apply new information. For example, features could be related to objects containing engineering tolerance information.

In modeling the assembly features of our components we have made no effort to give detailed geometric information about



(a) The conceptual graph describing some assembly features of the example brick in Figure 3.



(b) The type definition of the set of LEGO blocks.

Figure 5. Different views on the conceptual graph structure representing Lego assemblies.

the components or features. Instead we have encoded a relatively high level view of the basic shape of the features. However, it might be useful to be able to relate a component with detailed geometric information such as a solid model. More generally, it would be useful to be able to relate a component to other representations. This includes the solid models, STL/VRML files, natural language documentation, and 2D images.

Achieving this is simply a matter of adding to the supports for our conceptual graphs. A hierarchy of objects to describe other representations can be created and incorporated into **C**. These objects can include general information, such as a location where files can be retrieved, as well as more specific properties. For example, a transform can be included with an object containing information about a component's solid model to provide a mapping from the points used in the conceptual graph to those in the solid modeler. Such properties can be applied through type definitions such as the following:

```
type Representation(x) is
    [*x] → (locationURI) → [URI].
type GeometricRepresentation(x) is
    [Representation:*x] –
        → (transformation) → [AffineTransform].
```

Using these properties and others, hierarchies of component classes can be constructed for particular domains. Classes can define common properties for member components, such as particular features, as well as applying properties unique to that class. In our domain of LEGO pieces an obvious property particular to one class of pieces is that gears have a certain number of teeth, an important aspect about which we might want to be able to perform some reasoning. This fact can be incorporated into the domain through the following type definition:

```
type LEGOGear(x) is
    [LEGOElement:*x] –
```

```
    → (teeth) → [NonNegativeNumber].
```

Similarly, Figure 5(b) shows that all LEGO blocks must have certain features. These definitions can in turn be specialized by classes of components lower in the type hierarchy, as in the following definition of LEGO 24 tooth gears which prescribes a specific value for the *teeth* property as well as assigning a graphic representation for instances of that class:

```
type Gear24T(x) is
    [LEGOGear:*x] –
        → (teeth) → [NonNegativeNumber:24]
        → (representation) → [Image:pic156] –
                → (locationURI) → [URI:genid1110].
```

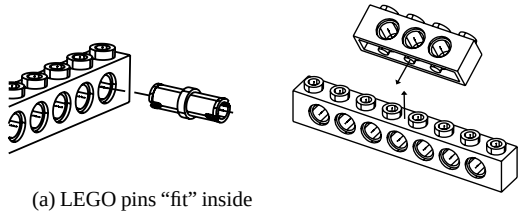
In contrast with modern CAD packages, our modeling of features avoids a detailed physical representation. Instead, we obtain detailed physical information through linking to other sources developed specifically for features and components. Abstract information that could be incorporated into this representation includes tolerance specifications, class specific information such as the number of teeth on a gear, color, cost, and supplier information. Many CAD systems include "attribute" entities or data fields to contain such information outside the realm of part geometry in an *ad hoc* fashion. This conceptual graph representation captures all data in a systematic fashion suitable for a variety of information to be modeled and reasoned upon.

Assemblies

With several important properties of components represented, we can now define properties of the *Assembly* class of objects. We will begin by adding another class to our hierarchy of types, *Part*, and denoting that assemblies as well as components are both parts: *Assembly* $\subset$ *Part* and *Component* $\subset$ *Part*. Another important assembly class is *RigidBodyAssembly*, a subclass of *Assembly*. This type is to be used to declare the assembly designer's intention that it is to be viewed as one solid part. Of course, one of the most important aspects of an assembly is that it contains components and subassemblies. As in the example of Figure 2(b) we will use the *part* relation to denote these elements.

Following this, an important aspect of assemblies are the relationships between their parts. Among these are static assembly constraints. In our domain of LEGO assemblies these consist mainly of "fits," "align," and "mate" relationships as in Figure 6. To represent these relationships we define classes of objects to describe the constraints and then relate the assembly object to instances of these. For example, the object to represent "fits" relationships is defined by the following conceptual graph:

```
type Fits(x) is
    [AssemblyRelation:*x] –
```



(a) LEGO pins "fit" inside holes in the bricks.

(b) Surfaces are "mated" to constrain attachment.

Figure 6. Static assembly relationships between LEGO components.

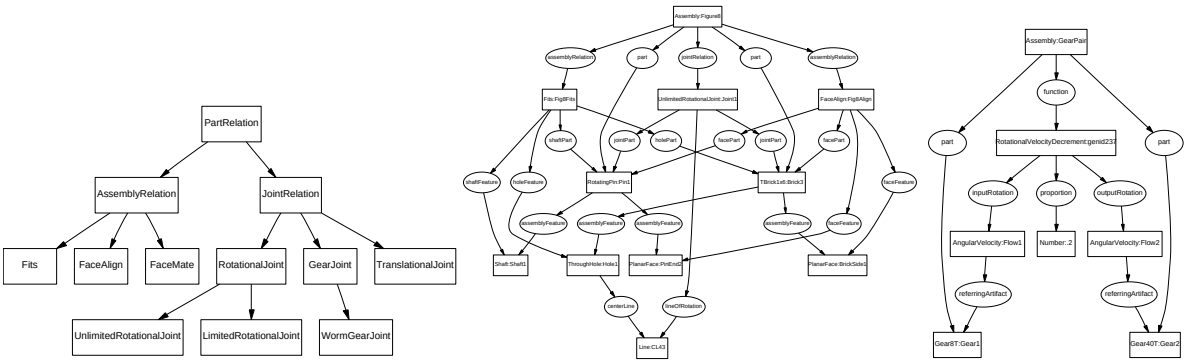
```
        → (holeFeature) → [Hole]
        → (holePart) → [Part]
        → (shaftFeature) → [Shaft]
        → (shaftPart) → [Part].
```

Besides static relationships, parts in an assembly may also have joints where motion interactions occur. In our domain these are mostly rotational, translational, and gear joints. Similar to the static relationships we define object classes to contain information about the joints and then relate assemblies to instances of these. The core pieces of information are the parts involved, and each specific type of joint incorporates additional information. For example, rotational joints have a line about which the parts rotate. These classes are defined as in the following:

```
type JointRelation(x) is
    [*x] –
        → (jointPart) → [Part:*p1]
        → (jointPart) → [Part:*p2].
type RotationalJoint(x) is
    [JointRelation:*x] –
        → (lineOfRotation) → [Line].
```

These definitions form the hierarchy of types shown in Figure 7(a). With these types incorporated into the conceptual graph supports, relationships between parts in an assembly can be modelled. Figure 7(b) shows such a modeling of the relevant static and joint relationships in pin and brick assembly of Figure 6(a).

As with features on components, our modeling of static and joint relationships reflects a small subset of those supported by modern CAD packages. However, the ontology of relations, as well as that of features, could be extended to incorporate more detailed information and classes. They could also be extended to incorporate elements from port based modeling as in (Sinha et al., 2001). Appropriate feature classes can be extended to include the information represented by ports. The relation objects could also be extended to include the information encapsulated by interaction nodes in that representation.



(a) Hierarchy of part relationship objects.

(b) Relationships between the brick and pin in Figure 6(a).

(c) Function and flow modeling of a gear pair.

Figure 7. Representation of assembly properties.

At a more abstract level of information than features and joint relationships, we can also represent the intended function of the parts. Classes of objects can be defined to represent functions and flows with instances associated to components and assemblies. Terminologies and taxonomies of functions and flows as presented in (Hirtz et al., 2001; Szykman et al., 1999a) can be directly incorporated into the type hierarchy. We can enrich the representation of these functions and flows through conceptual graph type definitions, formally defining properties of and constraints on these classes. Amongst others, range and cardinality constraints can be placed on relations associating flows with functions. Properties specific to certain classes can also be applied. For example, we can define a class of functions which decrement rotational velocities:

```
type RotationalVelocityDecrement(x) is
    [Decrement:*x] –
        → (inputRotation) → [AngularVelocity]
        → (outputRotation) → [AngularVelocity]
        → (proportion) → [Angle].
```

In this way the hierarchy can grow more specific and detailed while maintaining a well defined structure. Instances of these functions and flows can then be associated with assemblies and components through various relations. Such a function and flow modeling of a gear pair assembly is depicted in Figure 7(c).

This section has developed an ontology for describing engineering artifacts using conceptual graphs. Aspects from several existing representations have been incorporated, including CAD/detailed physical modeling, port based modeling, and function and flow. Incorporating other ontologies to extend the representation will require no revision of syntax or modification of the existing representation. Information at several levels of detail as well as application-specific information has and could be further incorporated, enabling a variety of queries, reasoning, and tools.

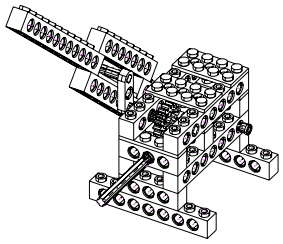


Figure 8. Mechanically different windshield wiper assembly.

Examples: Using the Representation

Figure 9 shows a conceptual graph representation of the windshield wiper assembly in Figure 1. This model identifies the mechanical joints present as well as important assembly relationships and the intended function of the assembly but omits many details. Of note is the blending of the function and flow ontology with the rest of the representation, ie. rotational mechanical joints are also rotational flows. Figure 8 shows a similar assembly with different mechanical properties. A representation of it is shown in Figure 10. This model is less detailed than the previous example but represents important aspects of the assembly including the mechanical joints and overall function.

With the assemblies represented in this form there are many interesting queries and operations which can be performed. One such activity is that of finding particular kinds of assemblies in

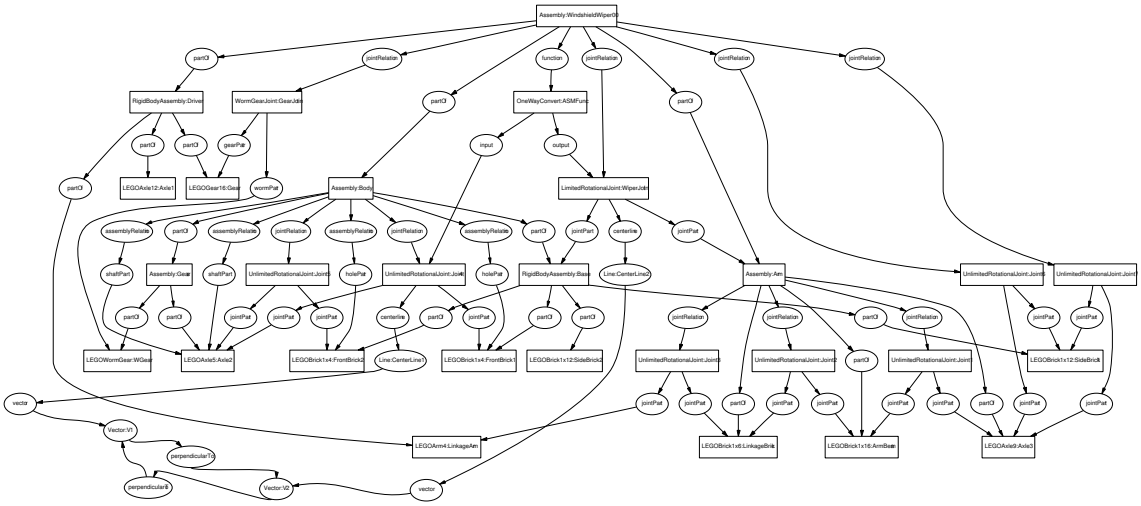


Figure 9. Conceptual graph representation of windshield wiper assembly in Figure 1.

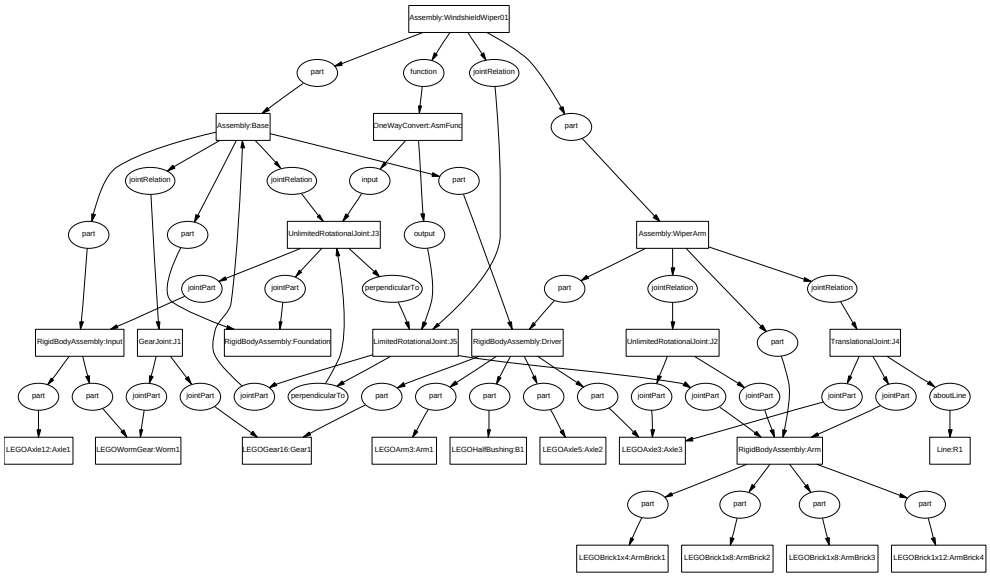


Figure 10. Conceptual graph representation of windshield wiper assembly in Figure 8.

a CAD database. We could perform searches to find assemblies containing particular joints and parts. However, a search based on the declared function of the assembly might generate less spurious results. To perform this query a conceptual graph can be used as a template to match against, as in Figure 11.

find `WindshieldWiper(x)` **as**

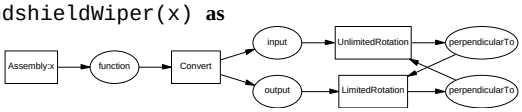


Figure 11. Query pattern for windshield wipers.

We can determine if an assembly in our database matches this query by determining if the query graph can be projected onto the assembly's model. One possibility for implementing this is to treat each element in the model as a fact and convert the query into a pattern of facts to unify against, as in the following:

```
Query WindshieldWiper(?x)
 ((type ?x Assembly)
  (function ?x ?y)
  (type ?y Convert)
  (input ?y ?z)
  (output ?y ?a)
  (type ?z UnlimitedRotation)
  (type ?a LimitedRotation)
  (perpendicularTo ?z ?a)
```

```
(perpendicularTo ?a ?z))
```

Note that although simple, this query incorporates aspects of abstract information, function and flow, as well as more detailed information related to the configuration of the assembly. In particular, this query stipulates that the two rotations must be perpendicular. The example of Figure 10 contains this information explicitly. However, in Figure 9 only the vectors for the centerlines of the rotation have been declared as perpendicular. This of course entails that the associated lines are perpendicular, which further entails that the rotations are perpendicular. A system performing such a query would have to include some sort of reasoning mechanism capable of performing that inference. As conceptual graphs are unsuited for such reasoning, some procedural hooks will have to be utilized to perform such queries.

We have restricted the representation presented here to *simple* conceptual graphs and their limits of expressive power. One aspect of these limits is that relations cannot be placed over subgraphs and therefore rules cannot be expressed. This capability would either have to be added or another mechanism used to carry out such inferences. For example, simple conceptual graphs can be bicolored through some means outside the content of the graph such that black elements represent precedents and white elements antecedents in a rule (Baget et al., 1999). These rules could, for example, then be converted into patterns of facts such as the previous example and used in a production system to carry out such inferences.

Implementation with DAML

This section discusses the use of the Resource Description Framework (W3C, 2000) to store instance data and the DARPA Agent Markup Language (DARPA, 2001b) to describe the artifact ontology. RDF is a language based on XML syntax for transcribing properties of objects. It is based on triples of *(subject, relation, object)* assertions. DAML is an RDF language of relations and objects used to describe ontologies—the elements and structure of a domain. In this discussion DAML refers to the March 2001 DAML specification (DARPA, 2001a).

Several other representations make use of XML syntax, including (Szykman et al., 1999b; Diaz-Calderon et al., 2000). The advantage of using XML is that it provides for a neutral basic format easily incorporated into many systems. However, it is still largely positioned at a syntactic level. The addition of new information into a data file may easily change the structure of the data such that although still valid XML, the system interpreting that structure will be unable to process it. RDF provides a more general structure which does not present this issue as directly.

Faced with such a general syntax and underlying data model, some mechanism is needed to describe the structure of the data. This should be done in a fashion based not on syntax, as is done in XML's Document Type Definitions, but rather on a more abstract structure of the information. DAML is one RDF-based language for achieving this. It defines relations such that we can describe the structure of the supports for our conceptual graph representation. This includes the type hierarchy and definitions as well as the relation terminology. For example, the previously defined *Point* class can be described in DAML as in Figure 12.

```
<daml:Class rdf:ID="&geom;#Point">
  <daml:intersectionOf parseType="daml:collection">
    <daml:Class rdf:about="&geom;#GeometricEntity" />

    <daml:Restriction>
      <daml:onProperty rdf:resource="&geom;#xcoord" />
      <daml:minCardinality>
        <xsd:nonNegativeInteger rdf:value="1" />
      </daml:minCardinality>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="&geom;#xcoord" />
      <daml:toClass rdf:resource="&units;#Distance" />
    </daml:Restriction>

    <daml:Restriction>
      <daml:onProperty rdf:resource="&geom;#ycoord" />
      <daml:minCardinality>
        <xsd:nonNegativeInteger rdf:value="1" />
      </daml:minCardinality>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="&geom;#ycoord" />
      <daml:toClass rdf:resource="&units;#Distance" />
    </daml:Restriction>

    <daml:Restriction>
      <daml:onProperty rdf:resource="&geom;#zcoord" />
      <daml:minCardinality>
        <xsd:nonNegativeInteger rdf:value="1" />
      </daml:minCardinality>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="&geom;#zcoord" />
      <daml:toClass rdf:resource="&units;#Distance" />
    </daml:Restriction>

  </daml:intersectionOf>
</daml:Class>
```

Figure 12. DAML description of the *Point* class.

It is worth nothing that even simple conceptual graphs are more expressive than DAML. For example, in a DAML-based application for finding models in a database it might be tempting to convert queries such as Figure 11 into DAML classes against which objects can be classified. However, this example query cannot be expressed in DAML because of the *perpendicularTo* relation. DAML has no provision in its language for describing these kinds of relationships between attribute values of a class. Although RDF data can take the form of a general graph structure, DAML can generally only describe data in tree form.

Despite this, much of our conceptual graphs' support can be described using DAML. The advantages of doing so are twofold: First, the structure of the domain is rendered in a formal and machine manipulable fashion. In this way tools and reasoners may make use of that knowledge in a way not possible if it were expressed using an XML DTD or other syntactic specification. This is also one of the goals of (Szykman et al., 1999b). Second, by making use of ontologies specified in DAML we can take advantage of the growing corpus of resources in and for that language. Beyond being able to use tools and software developed for DAML, we can also incorporate other ontologies to extend our representation. For example, more elaborate representations

of features or functions and flows could be incorporated.

To process data and knowledge in RDF and DAML form we have developed software (Kopena, 2001) based on Jess (Friedman-Hill, 1995), a Java-based production system. This software includes productions to process statements in the DAML language. For example, in the query of Figure 11 these rules would make the inference that the function *OneWayConvert* present in the assembly models is a subtype of the *Convert* function present in the query graph and can therefore be bound to the query. With this software we have begun to examine the application of our representation, initially to the problem given in the previous section of searching databases of models.

Discussion and Conclusions

This paper presented a holistic approach to representation of electromechanical assemblies based on conceptual graphs and implementaion with DAML. We believe this work describes a unique approach to integrating engineering design and knowledge representation. The use of conceptual graphs as an encoding scheme for diverse engineering knowledge creates a new framework in which assemblies can be defined and shared. By using DAML for our implementation, and developing engineering ontologies that are compatible with DAML, we have positioned our work to bridge the formal community building Semantic Web languages and the research community seeking coherent solutions to real-world problems requiring shared semantics.

Our examples show that our representation can be meaningfully applied to realistic objects. It should be noted that, while our examples are all taken from the Lego domain, nothing about our approach is limited to Lego-based devices. Additionally, we show that once information is formally encoded in our representation, interesting new questions can be posed and answered.

We see this work as illustrating a new methodology for representation and interrogation of design knowledge. With the goal of shared engineering semantics still distant, we hope this work shows a step along a new direction toward this important goal.

Acknowledgements. This work was supported in part by Office of Naval Research Grant N00014-01-1-0618.

REFERENCES

Anantha, R., Kramer, G. A., and Crawford, R. H. 1996. Assembly modelling by geometric constraint satisfaction. *Computer-Aided Design*, 28(9):707–722.

Baget, J.-F., Genest, D., and Mugnier, M.-L. 1999. Knowledge Acquisition with a Pure Graph-Based Knowledge Representation Model. In *Twelfth Workshop on Knowledge Acquisition, Modeling and Management*.

Bose, A., Gini, M., and Riley, D. 1997. A Case-Based Approach to Planar Linkage Design. *AIE*, 11(2):107–119.

DARPA 2001a. DAML+OIL March 2001 Specifications. `http://www.daml.org/2001/03/daml+oil-index`.

DARPA 2001b. DARPA Agent Markup Language (DAML). `http://www.daml.org/`.

Diaz-Calderon, A., Paredis, C., and Khosla, P. 2000. Organization and Selection of Reconfigurable Models. In *IEEE Winter Simulation Conference*.

Faltings, B. 1990. Qualitative Kinematics in Mechanisms. *Artificial Intelligence*, 44:89—119.

Forbus, K. 1984. Qualitative Process Theory. *AIJ*, 24:85.

Fowler, J. E. 1996. Variant Design for Mechanical Artifacts: A State-of-the-Art Survey. *Eng. with Computers*, 12:1–15.

Friedman-Hill, E. 1995. Jess: The Rule Engine for the Java Platform. `http://herzberg.ca.sandia.gov/jess/`.

Hirtz, J., Stone, R., McAdams, D., S, S., and Wood, K. 2001. Evolving a Functional Basis for Engineering Design. In *ASME* , DETC01/DTM-21688.

Hoffmann, C. M. and Joan-Arinyo, R. 1997. Symbolic Constraints in Constructive Geometric Constraint Solving. In *Journal of Symbolic Computation*, pp. 287–300.

Kim, G. J. 1997. Case-based Design for Assembly. *Computer Aided Design*, 29(7):497–506.

Kirschman, C., Fadel, G., and Jara-Almonte, C. 1996. Classifying Functions for Mechanical Design. In *ASME*, DETC96/DTM-1504.

Kopena, J. 2001. DAMLJessKB: DAML Knowledge Base Software. `http://plan.mcs.drexel.edu/projects/legorobots/design/software/DAMLJess%KB/`.

Kuipers, B. 1984. Commonsense Reasoning about Causality: Deriving Behavior from Structure. *AI*, 24:169—204.

Lee, K. and Andrews, G. 1985. Inference of the positions of components in an assembly: part 2. *CAD*, pp. 20–24.

Sinha, R., Paredis, C., and Khosla, P. 2001. Interaction Modeling in Systems Design. In *ASME*, DETC01/CIE.

Sowa, J. F. 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, Reading (MA).

Sowa, J. F. 2001. Proposed ISO Standard on Conc. Graphs. `http://users.bestweb.net/~sowa/cg/`.

Subramanian, D. and Wang, C. 1993. Kinematic synthesis with configuration spaces. In *Qual. Reasoning 93,*, pp. 228–239.

Szykman, S., Racz, J. W., and Sriram, R. D. 1999a. The Representation of Function in Computer-based Design. In *ASME*, DETC99/DTM-8742.

Szykman, S., Senfaute, J., and Sriram, R. D. 1999b. Using XML to Describe Function and Taxonomies in Computer-based Design. In *ASME*, DETC99/CIE-9025.

Umeda, Y. and Tomiyama, T. 1997. Functional Reasoning in Design. *IEEE Expert and Intelligent Systems*, 12(2):42–48.

W3C 2000. Resource Description Framework (RDF). `http://www.w3c.org/RDF`.