A FOUNDATION FOR INTEROPERABILITY IN NEXT-GENERATION
PRODUCT DEVELOPMENT SYSTEMS

Simon Szykman
National Institute of Standards and Technology
100 Bureau Dr., Stop 8260
Gaithersburg, MD 20899-8260

Steven J. Fenves
National Institute of Standards and Technology
100 Bureau Dr., Stop 8262
Gaithersburg, MD 20899-8262

Walid Keirouz
Department of Computer Science
Byblos Campus
Lebanese American University
475 Riverside Dr., #1845
New York, NY 10115

Steven B. Shooter
Department of Mechanical Engineering
Bucknell University
Lewisburg, PA 17837

Keywords: CAD, Information Modeling, Interoperability, Knowledge Representation Product Development

ABSTRACT
U.S. industry spends billions of dollars as a result of poor interoperability between computer-aided engineering software tools. While ongoing standards development efforts are attempting to address this problem in today’s tools, the more significant demand in next-generation tools will be for representations that allow information used or generated during various product development activities to feed forward and backward into others by way of direct electronic interchange. Although the next generation of tools has the potential for greatly increased benefits, there is also a potential for the cost of poor interoperability to multiply. The goal of this work is to develop representations of information that are unavailable in traditional CAD/CAM/CAE tools to support the exchange of product information in a distributed product development environment. This paper develops a vision of next-generation product development systems and provides a core representation for product development information on which future systems can be built.

1 INTRODUCTION
Since its advent, computer-aided design and manufacturing (CAD/CAM) has had an immeasurable impact on product development in engineering industry, and consequently on society as well. As a result of their significance, CAD/CAM technologies were identified by the National Academy of Engineering in 1989 (the NAE’s 25th anniversary) as an outstanding engineering achievement over the preceding 25 years, and more recently by the American Society of Mechanical Engineers as one of the greatest technologies of the 20th century (Petroski, 2000). As these technologies mature further, there is little doubt that their impact on product development will continue.

In the past, product development was often done within a single company by co-located design teams. In more recent years, there has been a shift in product development paradigms. Product development is more often done by geographically and temporally distributed design teams. There is a high level of outsourcing, not only of manufacturing but of actual product development efforts. Product development across companies, and even within a single company, is often done within a heterogeneous software tool environment. The Internet and intranets are supplanting paper and telephones as a means of exchanging product development information. As a result of this new product development paradigm, there is a greater need for software tools to effectively support the formal representation, capture, and exchange of product development information.

* Senior Research Associate, University Professor Emeritus of Civil and Environmental Engineering, Carnegie Mellon University.
† This work was performed while Professors Keirouz and Shooter were guest researchers at the National Institute of Standards and Technology.
The existing generation of computer-aided engineering (CAE) software tools has undeniably revolutionized product development in contrast to methods used before the advent of these technologies. Nevertheless, the current generation of product development software tools addresses the needs of traditional product development processes, and does not adequately support the needs of industry’s new paradigm described above. People are exchanging information across distributed design teams and corporate boundaries earlier, and reusing information to a greater extent. But because existing software tools do not capture a broad spectrum of product development information, these exchanges occur informally (face-to-face across a table, by phone, by paper). It is a lack of formal representations for product development information that creates a significant barrier to its effective capture and exchange.

In other words, engineers are getting by not with the support of existing software tools but despite the lack of support from them. The current generation of software tools is not designed to support at a formal level the kinds interactions that occur to the extent that engineers would prefer. These views are not held exclusively by the authors, but are representative of industry input obtained at several workshops held at the National Institute of Standards and Technology (NIST) in recent years: The NIST Design Repository Workshop (Szykman et al., 1998), Tools and Technologies for Distributed and Collaborative Design Workshop, and the NIST/ATP Workshop on Intelligent and Distributed CAD (Mitchell and Szykman, 1998).

The CAD/CAM/CAE software industry is ultimately a customer-driven one. It is therefore expected that as needs (such as those identified previously) mount, a new generation of tools will emerge to address these needs. The question that remains unanswered is: at what cost? Despite the acknowledged benefits of today’s tools, their widespread use has in many cases come at great expense to industry as a whole because of a lack of interoperability between tools. While these expenses do not approach the level of savings resulting from the use of CAE technologies, they do offset the overall benefits. These expenses are inevitably passed on to industry customers, and ultimately to consumers. A study performed by the NIST Strategic Planning and Economic Assessment Office conservatively estimates the economic cost due to lack of interoperability in the automotive supply chain alone at one billion dollars per year (NIST, 1999). The study also estimates similar interoperability costs in the aerospace, shipbuilding, and construction machinery industries at about 400 million dollars per year.

As the complexity of products increases and product development becomes more distributed, new software tools will begin to cover a broader spectrum of product development activities than do the traditional mechanical CAD systems. Accordingly, the ability to effectively and formally capture additional types of information will become a critical issue. This paper develops a vision of next-generation product development systems and provides a core representation for product development information on which future systems can be built. This paper does not put forth specific technologies that will be incorporated into next-generation tools, but rather seeks to address potential interoperability problems proactively, rather than reactively, by providing this core as a foundation for improved interoperability among software tools in the future. The costs of poor interoperability among today’s tools are likely to be significantly compounded in the future if the problem remains unaddressed. The work presented in this paper addresses a fundamental problem whose solution can impact literally billions of dollars of costs to industry.

The economic benefits notwithstanding, the effort of developing a generic knowledge infrastructure for the next generation of tools is one that neither industry nor the CAD/CAM/CAE vendor community is likely to undertake alone. Historically, software tool vendors have considered proprietary data representations—a significant source of interoperability problems—as part of their competitive advantage. Working to eliminate the barriers to interoperability is often viewed by a software vendor as something that will make it easier for customers to purchase and use a competitor’s products rather than those sold by that company. This perceived threat to a competitive advantage provides a disincentive for vendors to address interoperability problems. Indeed, this is one of the main reasons that efforts to integrate data exchange standards such as STEP (Standard for the Exchange of Product Model Data, ISO 10303) (ISO, 1994) into the existing generation of software tools has met with resistance from vendors despite requests from, and clear benefits to, their industry customers (Szykman et al., 1998).

While engineering companies in industry stand to benefit from work along these lines, they too are unlikely to undertake such an effort. This is a result of industry research and development trends that have shifted from basic R&D to project-oriented R&D. This change makes it difficult for a company to use project-specific budgets to fund work that is acknowledged as beneficial, but has diffused impact. What limited funds are available for generic R&D are generally applied to problems where benefits can readily be mapped either to a set of related projects, or to the corporate bottom line, in a short time frame.

These cultural issues have served as obstacles to the development of technical solutions to interoperability problems in existing tools, and run the risk of similarly affecting the next generation of software systems. The National Institute of Standards and Technology has U.S. industry as its primary customer and works to address problems that have significance to industry but that companies are not likely to solve on their own for one reason or another. NIST’s emphasis is on economic impact to industry and society on a broad level rather than a corporate bottom line. Furthermore, NIST is not biased toward a particular class of problems, company, or industry sector, focusing instead on generic solutions that have broad-based applicability in industry. As a result, NIST is uniquely situated to invest in an effort to anticipate and address interoperability needs in next-generation product development systems.

This paper is organized as follows: Section 2 describes related work done in the area of representation and modeling of design knowledge, Section 3 discusses interoperability issues in
the context of product development, Section 4 presents a vision for next-generation product development systems, Section 5 presents a representation for product development information that addresses the needs motivated in earlier sections, and Section 6 provides a discussion of areas for future research.

2 RELATED RESEARCH

Traditional CAD systems are limited to representation of geometric data and other types of information relating to geometry such as constraints, parametric information, features, and so on. The engineering design community has been developing new classes of tools to support knowledge-based design, product data management (PDM), and concurrent engineering. When contrasted with traditional CAD tools, these new systems are making progress toward the next generation of engineering design support tools. However, these systems have been focusing primarily on database-related issues and do not place a primary emphasis on information models for artifact representation. This work is most directly descended from the representation developed as part of the NIST Design Repository project (Zytko et al., 2000). That work is based in part on the CONGEN architecture (Gorti et al., 1998), which made use of the SHARED object model (Wong and Sriram, 1993) as a representational foundation. As will be described in Section 5, the work presented in this paper drew not only on the NIST Design Repository project, but also several other ongoing efforts at NIST.

3 INTEROPERABILITY AND PRODUCT DEVELOPMENT

As cited previously, U.S. industry spends billions of dollars as a result of poor interoperability between computer-aided engineering software tools. Ongoing standards development efforts in various industries are attempting to address this problem. Among the success stories in standards development is STEP (Standard for the Exchange of Product Model Data). While individual companies hesitate to provide actual costs associated with specific processes or activities, they have been willing to provide a few data points regarding the use of STEP. The Lockheed Martin F-22 program shows consistent savings associated with specific processes or activities, they have been willing to provide a few data points regarding the use of STEP. The Lockheed Martin F-22 program shows consistent savings using STEP, including a 50% process savings for composites and a projected savings of 27% on tool design for CAD/CAM systems. The Boeing 767 and 777 programs reported a 75% time savings in processing designs from engine suppliers using STEP, and the Boeing C-17 program reduced time to transfer bill of materials data from weeks to minutes using STEP, with an average of 8000 part data exchanges per night (PDES, 1999).

While these data indicate the utility of data exchange standards to improve interoperability, viewing these as cost savings is the result of a perspective skewed by the context of poor interoperability. This level of savings can only be achieved because the costs wasted by interoperability problems are very high. In other words, the remedy to the problem is of great importance only because the problem itself is so significant. Faced with these problems today, a retrospective viewpoint leads to the observation that if interoperability issues had been successfully addressed earlier on, industry might have avoided some of these wastes costs entirely, instead of working to reduce waste after the fact. It is this observation, along with a projection of how product development in industry is changing, that motivates the need to address these issues in next-generation product development software systems.

While existing standards efforts focus on enabling interoperability among tools that address a given product development activity (such as geometric CAD), the more significant demand in next-generation tools will be for representations that allow information used or generated in various product development activities to feed forward and backward into others by way of direct electronic interchange. The objectives of the present work are to develop representations of information that are un-
available in traditional CAD/CAM/CAE tools, to support exchange of knowledge in the new product development paradigm, and to help avoid a proliferation of proprietary formats in next-generation commercial software tools. This work is not itself a standards development effort. Rather, it is an attempt to identify needs and provide a generic information representation core that can serve as a foundation for development of new systems, and at some point for future standards development efforts. We do not propose technologies that affect implementation-level development of software tools, but provide an infrastructure that will facilitate the capture and exchange of information among commercial tools that use it as a starting point and build upon it.

4 NEXT-GENERATION PRODUCT DEVELOPMENT SYSTEMS

The basic assumption underlying this work is that the change in product development paradigm identified in the introduction will lead to a new generation of software tools. This next generation of tools will be driven by an increased industry focus on representation, capture, and exchange of product information and product development knowledge. Existing computer-aided engineering tools are focused primarily around geometric CAD and analysis, with some tools supporting CAM activities such as process planning. Next generation systems will enable the capture of a broader variety of product information, and will support a wider range of product development activities than do existing tools.

The vision held by some for future product development tools is that of a monolithic software system. In this vision, the product development process will be supported by a single integrated application suite. Such a tool would attempt to address the needs of the new product development paradigm, allowing teams that are potentially distributed geographically or across corporate boundaries to access tools and data at different phases of product development in order to produce a product (see Figure 1).

This vision, though not an uncommon one, has a number of drawbacks associated with it. In general, since a monolithic system is intended to be as complete a solution as possible, less emphasis is put on interoperability with other systems. Because interoperability becomes a problem, collaboration with users of other tools becomes difficult. What has happened in some cases among the existing generation of tools is that a large company will make a statement to the effect of: “If you want to do business as part of our supply chain, you will use this particular software tool.” Since monolithic systems tend to be expensive, among those priced out of the running are many small and medium sized businesses that form a large segment of the industry community. These limitations stifle competition, resulting in prices to the customers and end users that are higher than they might otherwise have been. Such systems also tend to tie users

![Figure 1. Monolithic product development system.](image)
to one vendor, making the migration to a new tool suite problematic should a better system come along at some later point in time. And lastly, a monolithic tool suite will often result in a sub-optimal set of technologies. It is unlikely that a system which provides technologies for a variety of activities will be the best at what it does in all cases, particularly when a solution that might be best solution for one company may not be for another.

Some software vendors, rather than working to build a monolithic system, are working with a business model under which they establish relationships with other software companies and work to design interfaces between tools to make integration more seamless. This philosophy has the potential to offer greater customization of software tool suites. The drawback, as a practical matter, is that choice is likely to remain limited, this time to a set of companies who have established these strategic partnerships.

In our view, the ideal next-generation systems for product development will be those with which individual companies, or teams involved in given product development activities, can collaborate using a heterogeneous set of software tools, and still meaningfully exchange information and pass knowledge between various phases in the process. Assuming the interoperability barrier can be overcome, this vision avoids several of the disadvantages associated with a monolithic system. Companies would not be required to standardize on the same software platform in order to collaborate on product development. Smaller companies using individual software tools due to limited resources would still be able to compete to be part of a larger company’s supply chain. Larger companies would be able to assemble what they consider to be the best suite of tools from a selection of existing software products (possibly from competing vendors), and would be able to migrate more easily to new tools, although some effort would still be involved in doing so.

We define the Engineering Context as the entire body of information and engineering knowledge that evolves throughout the product development process. Under the vision of a customizable and flexible product development tool suite, multiple software tools—potentially from competing vendors—are used during different product development activities at various stages of the design process, creating, adding to, and modifying the engineering context. Information generated by some tools during some activities will be used by other tools at other stages of the design process. In order to achieve the interoperability required by this vision, a tool that requires information created by a different tool must be able to receive that information irrespective of which tool created it. This, in turn, requires that the formal representation of product information (or at least that portion of the engineering context that is to be shared with other applications) be available in a standardized form. It is important to note that tools need not be able to read native data files created by other systems; what is required is only that a tool be able to serve information in a format that other tools can interpret. Thus, although vendors would need to agree to use this standardized format for exchange of engineering information, they would not be required to standardize on native or application-internal data formats.

The vision of a flexible product development tool suite is shown in Figure 2. In some cases, tools from a single vendor may be designed to share a common database as illustrated with the users in the top half of the figure. In other cases a single tool may have an individual database as shown in the bottom left portion of the figure. The “X”es in the figure denote the data exchange interfaces required to exchange or externally store product information in a standardized format. Although it is not necessary for vendors to adopt the standardized representation as their native representation, those that do would not require a translation step to exchange or externally store data. The software tool in the bottom left portion of the figure does not show a data exchange interface for this reason.

The large database at the bottom of Figure 2 represents the engineering context. The database is shown as being in a cloud, to indicate that the single database shown is merely representative of this body of information rather than being indicative of its true instantiation. Whether the actual instantiation of the engineering context takes the form of a centralized database, a distributed database, a collection of individual files created by separate applications, etc., is an implementation-level issue. The focus of this work is not on implementation, but on providing a generic core for representing the engineering context as a basic foundation for interoperability. Regardless of the form that the engineering context embodies, for a tool to be able to retrieve and interpret information generated by other tools in a heterogeneous software environment, interoperability issues must first be addressed at the information representation and exchange level.

### 5 DEFINING THE ENGINEERING CONTEXT

#### 5.1 Technical Approach

The preceding discussion presents a vision that motivates the development of a foundation for representation of design information that is non-proprietary and not tied to a single vendor. From the interoperability perspective, in an ideal world all vendors would use the same representation making interoperability problems irrelevant. Realistically, it is not likely that all CAD/CAM/CAE vendors will agree on a common product information representation, or even that they will want seamless exchange with competing systems. Nevertheless, work toward this goal is still very valuable for several reasons. First, agreement on requirements and implementations to any extent is better than none. To whatever degree a specification is adopted in multiple implementations, exchange of information will be that much less problematic even if a total exchange is not seamless. Addressing the problem in anticipation of next-generation systems, rather than in response to failings among those systems after have been developed, increases the likelihood that common solutions will be adopted. Second, assuming realistically that some interoperability problems will exist in the future, it is also probable that some sort of standards develop-
ment efforts will ensue in response. The work being done here can provide a starting point for those future standards.

For both of these reasons, simplicity is a key requirement for the representation being developed. Simplicity also makes a proposed representation more appealing to users. This is a critical issue since for vendors to even consider such a representation, they must believe that a broad market will exist for tools built upon it. Because of a need for broad appeal among potential customers, a product information representation should be domain-independent and should not be tied to any one product development process.

To address this last point, an effort in modeling design information flow was undertaken. This effort examined several product development processes and proceeded to develop a generic model of the flow of product information independent of any one design process model. This project identified common classes of information used by designers, activities that were common to various design processes (even though the sequences of those activities were often not common to different processes), as well as the kinds of abstractions and transformations of information that occurred in the product development process. The results of that effort, described further in (Shooter et al., 2000), provided significant input into developing the actual content requirements for the information representation presented in this paper.

Based on these issues, in order to enable the vision of next-generation product development discussed earlier, the goal that was set for this effort was to develop formal representation that could form the core of the engineering context that was discussed previously above. This core is intended to be an abstraction of concepts that are common across many design activities. The broader engineering context would include this common core along with additional concepts (objects, relationships, attributes, etc.) such as activity- and domain-specific extensions and specializations that may not be common to many activities. The core would provide interoperability across activities, while extensions to the core might or might not require additional translators or interpreters depending on the extent of adoption among different software tools. The requirements for the core were to develop a knowledge representation that is:

- not tied to a single vendor software solution,
- open and non-proprietary,
- simple and generic,
- expandable by allowing augmentation of the core with additional concepts to create a broader engineering context,
- not dependent on any one product development process, and
- capable of capturing that portion of the engineering context that is most commonly shared in product development activities.
The representation was also intended to provide interoperability among four in-house research and development projects: (1) the NIST Design Repository project, developing a distributed framework to support the creation of design repositories, the next generation of design databases (Szykman et al., 2000); (2) the Design-Process Planning Integration project, developing interface specifications and prototypes to enable manufacturability analysis during conceptual product design (Feng et al., 1999); (3) the Design for Tolerancing of Electro-Mechanical Assemblies project, working to advance the use of tolerancing at early stages of design and to investigate the best use of available methods of tolerance analysis and synthesis (Roy et al., 1999); and (4) the Object-Oriented Distributed Design Environment project, developing a software prototype of such a system.

Each project team had been proceeding to develop its own product model, with little or no interaction among the projects. A comparison of the product models developed or proposed by the four projects showed that of the 133 terms (object and attribute names) used in the models, 99 terms (74%) appeared in one model only and only 3 (2%) appeared in all four models. This comparison excludes terms that are specific to the domain of one project only, such as process- and tolerance-related terms. It therefore appeared that there was not much commonality.

The next step was the development of a core combined product model that could be shared across projects and extended as needed to suit the concerns of the individual projects. Although the core model most closely resembles the model originating from the NIST Design Repository project, terms from the other three projects were also incorporated, showing the synergism provided by broader exposure and discussion. The examination of multiple independently-developed models, abstracting out the commonalities, and distilling their basic information content, led to a more generic and extensible representation than any one of them had previously provided. The representation that emerged is proposed as an information transfer mechanism for next-generation product development tools, either in the basic form presented in this paper or as the “foundation” or base-level representation of the multilevel design information flow model presented in (Shooter et al., 2000).

5.2 The Core Representation

The core representation consists of classes of objects and relationships. The full listing of all classes in the core representation is shown in Appendix A. In the text that follows, names of classes are capitalized and names of attributes are not. The syntax and common characteristics of the classes are discussed first, followed by their semantics.

The object class hierarchy is shown in Figure 3. An "(a)" denotes an abstract class, which is a class for which instances cannot be created, but which exists for the convenience of grouping attributes that are common to all of its subclasses so that these attributes can be inherited by the subclasses. Syntactically, all object classes are specializations of the abstract class DRP_Object which has attributes name, information, and sources and destinations of references and relationships (discussed below). The abstract class Restricted_DRP_Object specializes the DRP_Object class and has additional attributes has-constraint and is_required_by serving as links to Constraints and Requirements, respectively. The specializations of the DRP_Object class are the Artifact, Behavior and Specification classes; specializations of the abstract class Restricted_DRP_Object are the Function, Flow, Form, Geometry and Behavior classes. The Function class further specializes into the Transfer_Function class. The attributes of the object classes are discussed below in the context of their semantics.

The relationship class hierarchy is shown in Figure 4. Similarly to objects, all relationships are specializations of the abstract class DRP_Relationship, with attributes name and information. The specializations are the Requirement, Reference, Constraint and Set_Relationship classes, the latter further specializing into Directed_Set_Relationship and Undirected_Set_Relationship classes.

In order to make the representation as robust as possible without having to predefine all possible attributes that might be

![Figure 3. Object class hierarchy.](image-url)
relevant in a given set of domains, the core representation is limited to those attributes required to capture generic types of product information and to create links and associations among the entities shown in Figures 3 and 4. The representation intentionally excludes attributes that are domain-specific or entity type-specific. Such attributes can be represented, but are not explicitly built into the schema. Instead, each object and relationship has an Information entity associated with it. The class Information is a container for all detailed information, and consists of a brief textual description, a textual documentation string (which typically provides a file path or URL referencing more substantial documentation than is given by the brief description), a methods slot for the methods operating on the object, and a properties slot that contains a set of attribute-value pairs stored as strings. Domain- or entity type-specific attributes are represented here as properties. This lack of specialization of entities results in a small number of entity definitions that are broadly applicable. As will be illustrated in an example below, this allows the representation to be used to represent a 9-Volt electric motor and a 9-Volt electrical current flow without predefining a domain-specific attribute called voltage in either the artifact or flow entity definitions.

In order to make the representation as robust as possible without having to predefine all possible attributes that might be relevant in a given set of domains, the core representation is limited to those attributes required to capture generic types of product information and to create links and associations among the entities shown in Figures 3 and 4. The representation intentionally excludes attributes that are domain-specific or entity type-specific. Such attributes can be represented, but are not explicitly built into the schema. Instead, each object and relationship has an Information entity associated with it. The class Information is a container for all detailed information, consisting of a brief textual description, a textual documentation string (which typically provides a file path or URL referencing more substantial documentation than is given by the brief description), a methods slot for the methods operating on the object, and a properties slot that contains a set of attribute-value pairs stored as strings. Domain- or entity type-specific attributes are represented using the properties attribute. This lack of specialization of entities results in a small number of entity definitions that are broadly applicable. As will be illustrated in an example below, this allows the representation to be used to represent a 9-Volt electric motor and a 9-Volt electrical current flow without predefining a domain-specific attribute called voltage in either the artifact or flow entity definitions.

All specializations of the abstract class DRP_Object except Flow have their own decomposition hierarchies, represented by attributes such as subartifacts/subartifact_of for the Artifact class. In addition, most of the objects and relationships also have an attribute called type, a symbolic classifier. All of the entity classes that have a type also have an individual hierarchical taxonomy of terms associated with them. These terms are generic enough to designate a broad variety of engineering artifacts, and yet concise enough to provide a manageable standardized vocabulary. The type classifier is a string corresponding to one of the terms within a given taxonomy. For example, the engineering function “Convert” is one of numerous subtypes of “Transform-function” which is one of several subtypes of “Function”. Another way to view this is that several of the entities in the representation have their own individual generic engineering classification hierarchies that are unrelated to the entity class hierarchies shown in Figures 3 and 4.

The motivation for providing typing information and associated taxonomies on which to draw for artifact modeling is twofold. The standardized vocabulary facilitates indexing and retrieval of product knowledge for design reuse. The classification of an entity (e.g. an Artifact or Function or Flow) within its associated taxonomy allows a software system to reason about these entities based on knowledge of their type. In other words, knowing that an entity is an Artifact gives you information at one level, but knowing that an artifact is a motor allows more sophisticated reasoning. The type attribute is provided as part of the infrastructure for product information representation with the expectation that it could be used in these ways in future software tools. A more substantial discussion of taxonomical issues and examples of taxonomies for engineering function and flows are given in (Szykman et al., 1999). Discussion of the mechanisms for indexing, retrieval, and reasoning based on type knowledge in future systems are beyond the scope of this paper.
Turning to semantics, the key object class is the Artifact. It represents a distinct entity in the design, whether that entity is a feature, component, product, subassembly or assembly. All such entities can be represented and interrelated through the subartifact/subartifact_of containment hierarchy. The artifact’s attributes, other than the common ones described above, refer to the Specification responsible for the Artifact and the Form, Function and Behavior objects constituting the Artifact. A Specification is an object which contains information relevant to an artifact that is derived from customer needs. An artifact Specification drives requirements among one or more other Function, Form, Geometry, Material and Flow entities via a requirement relationship (to be discussed below). An additional attribute, Config_Info, links the Artifact to an element of the class Config_Info that represents design process-related attributes of the artifact, such as state, level (as used in Shooter et al., 2000), version, etc. in an interactive environment.

The Form, Function and Behavior classes represent the traditional components of design representations that were discussed in Section 2. A specialization of the Function class is the Transfer_Function class, which represents a transformation between one or more flows (e.g., current, liquid, energy) and explicitly refers to the Flow entities involved using input_flow and output_flow. The Flow class identifies the flows involved and references the Artifacts corresponding to a flow’s source and destination. The Form class refers to the two principal aspects of the form, namely the artifact’s Material and Geometry. This subdivision was introduced into the core model because some of the intended applications, such as the Design-Process Planning Integration and the Design for Tolerancing projects tend to treat these two aspects quite differently (e.g., the task of material selection for a given function and geometry in process planning).

A Requirement entity is a one-to-many relationship between a Specification and a set of Function, Form, Geometry, Material and Flow entities governed or otherwise affected by that specification. Similarly, a Constraint entity is viewed as an undirected set membership relation among the constrained Function, Form, Geometry, Material and Flow entities. If it is intended to represent a mathematical equality or inequality constraint, the properties slot of the constraint can store the attributes entering in the constraint as well as the relational operator. The Undirected_Set_Relationship class simply sets up set membership relationships among objects, while the Directed_Set_Relationship class identifies a special member of each set. Finally, since, as stated above, the representation allows for hierarchies of Function, Form, Geometry, Material entities independent of each other and of the artifact’s containment hierarchies, it was found prudent to introduce the Reference class of one-to-one directed relationships between referring and referred_to objects as a means of navigating between elements of such hierarchies.

As can be seen from the above discussion, the core representation is by no means minimal. The set of object classes largely reflect traditional terms used in formal design descriptions and models; it was felt that any further abstraction would have eliminated semantically meaningful terms. The number of relationship classes could have been reduced since requirements or constraints could have been represented using the more generic set relationships. However, it was felt that the terms “requirement” and “constraint” are themselves semantically meaningful and should be retained.

5.3 Example: Power Drill Motor

This section presents an electric motor that is a component in a power drill as an example to illustrate the use of the product knowledge representation core. What follows is a set of instances of several of the entity classes described above. The intent of this example is to provide a flavor of how the simple generic entities that comprise the representation core can be used to model a complex engineering product, as well as what the entities that populate a product information base look like. This section does not attempt to provide a complete representation of the power drill motor (as is evidenced from the fact that the entities shown contain pointers to various other entities that do not appear in the example), nor does it attempt to provide a comprehensive set of examples that covers every kind of entity discussed above. Note that although entities in an artifact model always contain all the attributes defined for that entity (see Appendix A) irrespective of their value, in this example attributes having default or NULL values have been omitted for reasons of brevity.

The first entity below is an Artifact representing the drill motor. This motor is a subartifact_of another Artifact representing a power transmission (not shown), which has several other subartifacts in addition to the motor, including a chuck, gearbox, etc. The power transmission itself is one of several subartifacts of the power drill itself. It is in this way that the physical decomposition of an engineering product is represented. The drill motor itself does not have subartifacts, indicating that it is not further decomposed into subassemblies and components. The reason for this is that although the motor is comprised of multiple parts, the company that manufactures the drill buys the motors as OEM catalog parts, and thus treats the motor as a single component. Were this not true, there is nothing to prevent the representation of each of the motor components as subartifacts of the motor.

Artifact
{
  name Drill_motor
  information Drill_motor_info
  type Motor
  config_info Drill_motor_config
  function Drill_motor_function
  form Drill_motor_form
  behavior Drill_motor_behavior
  subartifact_of Power_transmission
  is_source_of (Motor_rotation)
  is_destination_of (Battery_current)
}
The motor shown above includes a reference to a Function object, providing a pointer from the artifact domain into the function domain. The next entity represents the motor Function, which is to "convert". The input_flow and output_flow to the Function are an electrical current and rotational motion respectively, shown in the entities that follow. Each of the Flows has pointers to a source and destination, which are Artifacts, thereby providing pointers back from the function domain to the artifact domain. The information captured in aggregate is that the motor has a function, which is to convert electrical energy that flows from the battery pack connector cable to the motor, into rotational motion whose source is the motor and whose destination is a gearbox.

Transfer_Function
{
  name Drill_motor_function
  information Drill_motor_function_info
  type Convert
  function_of_artifact Drill_motor
  input_flow {Battery_current}
  output_flow {Motor_rotation}
}

Flow
{
  name Battery_current
  information Battery_current_info
  has_constraint Battery_current_constraint
  type Current
  source Battery_pack_connector_cable
  destination Drill_motor
  is_input_of (Drill_motor_function)
  is_output_of (Battery_pack_connector_cable_function)
}

Information
{
  name Battery_current_info
  properties {"V = 9 Volts"}
}

The next two entities below represent a Constraint that is applied to the battery current requiring that the electrical input to the motor should be at 9 Volts. This previous entity above represents a property of the battery current, which indicates that the voltage of the battery is 9 Volts. Thus, the constraint is satisfied.

Constraint
{
  name Battery_current_constraint
  information Battery_current_constraint_information
  type plaintext
  constrains Battery_current
}

5.4 Implementation

The product representation core presented in this paper is being adopted as the basis for software tool implementations in the Manufacturing Systems Integration Division (MSID) at NIST. The NIST Design Repository project, geared toward providing a technical foundation for the creation of heterogeneous information repositories that support the representation, capture, sharing, and reuse of corporate design knowledge is an example of such an effort. This project has developed a prototype implementation that includes web-based interfaces for authoring, updating and modifying design repositories (Szykman et al., 2000). As part of this ongoing effort, a second prototype implementation is currently in progress.

The NIST Design Repository project has created several repositories of design information, including artifact models for an ultra-high vacuum transport system, a cordless power drill, a
hand saw, and a detail sander. Translators are being developed to convert these artifact repositories to the product representation core presented in this paper. Once the second prototype implementation is complete, these repositories along with any new ones subsequently created will be available to demonstrate the utility of the product information core.

In addition to the NIST Design Repository project, three other projects in MSID involving tolerance design, assembly design, and design-process planning integration have tentatively agreed to make use of the representation core where suitable. When those projects reach appropriate levels of implementation, it will be possible to utilize the core not only for artifact modeling, but also for sharing and exchange of product information via interactions between projects.

As was illustrated in the example in the previous section, the existing representational structure allows individual behavior models to be attached to various parts of a product. Other researchers are developing methods for developing generic interfaces between multi-domain simulation models to allow composable simulations that would automatically assemble individual component models into a system-level simulation. Allowing system-level rather than just component-level simulation would clearly improve a designer’s ability to evaluate or validate a design. We are currently engaged in discussions with researchers at Carnegie Mellon University to explore the possibility of demonstrating the use of these types of models via implementation (Diaz-Calderon et al., 2000).

6 AREAS FOR FUTURE RESEARCH

This paper presents a core representation for design information that is intended to serve as a representational foundation for next-generation product development systems. The work is motivated by the high costs of poor interoperability in the current generation of CAD/CAM/CAE software tools. These costs are currently on the order of billions of dollars, and threaten to grow further as functionality of tools increases and their usage extends beyond traditional geometry-based design activities. The research that led to the development of this core representation drew high-level needs from a vision of next-generation product development systems, drew specific content-level requirements from a related effort in design information flow modeling, and was synthesized after an analysis of several independently-developed design artifact representations.

Although this paper does not address the specific technologies that will be incorporated into next-generation tools, this research does attempt to provide a foundation for those systems by providing a simple and generic representation infrastructure. Specifically, automated reasoning will be easier using formally-represented product information than informal or unstructured information such as text-based documentation. The structure of product information that is captured, such as physical decomposition, functional decomposition, and mappings between the two, will further support reasoning about designs. The development of multiple taxonomies of engineering terminology (an ongoing activity in the NIST Design Repository project) for use with the type classification attribute will greatly facilitate the indexing and retrieval of product information from engineering databases. Again, the actual development of mechanisms for indexing and retrieval of product databases is an area for future research.

Also related to the processing of information in the core representation is the treatment of an object’s attributes as strings containing attribute-value pairs in the property slot. This approach stemmed from the initial objective of the NIST Design Repository project, which originally dealt with the storage and retrieval of archival information rather than interactive product development. The benefit of this method is the generic nature of the object class hierarchy that allows the representation of a broad number of concepts without specialization into a large number of subclasses. The lack of specialization of entities may be a benefit at the representation level in terms of simplicity, but may turn out to be less suitable as a mode of information transfer for tools where rapid and seamless interoperability is needed, as in the design environment sketched in Figure 2. The use of the core schema in such environments needs to be investigated further. One promising approach to examine is to treat the core as a “virtual global schema” and append to its objects domain-specific attributes and additional objects and relations which need to be known only by the tools or agents actually involved in transactions.

In the near term, implementation efforts in the NIST Design Repository project (described in Section 5.4) will continue toward the development of a second prototype built upon the representation core. In this implementation, the engineering context is stored in a single relational database. At the representation level, there is nothing constraining the engineering context to this form; in practice the engineering context may be a centralized database, a distributed database base, or possibly a “virtual” knowledge base that resides within the databases of several different software applications.

The final significant area of future work is to gather feedback and buy-in from the community of both researchers and end-users. Researchers at several universities are contributing to the NIST Design Repository project or making use of representations developed for the project. In addition to continuing these interactions, NIST will be holding an industry workshop in the Summer of 2000 to gather additional information regarding needs associated with interoperability in next-generation product development systems. The intent of this workshop is to bring together both software vendors and industry end users to obtain feedback on work to date as well as input into future activities.

ACKNOWLEDGMENTS

The authors would like to thank Jocelyn Senfaute for his comments on the product knowledge representation core, and for aiding in refining and maintaining the information models presented in this paper.
REFERENCES


APPENDIX A: THE PRODUCT KNOWLEDGE REPRESENTATION CORE

Notes:
• An abstract class is a class for which instances cannot be created, but which exists for the convenience of grouping attributes that are common to all of its subclasses so that these attributes can be inherited by the subclasses.
• As described in Section 5.2, the type attribute for objects and relationships is a string that is required to be one of the terms within a taxonomy associated with that kind of entity. Abstract classes do not have types since instances for abstract classes do not exist.
• In addition to the entity definitions shown below, a separate set of entities also exists for the organization of terms into taxonomies used for entity type classification. As the focus of this paper is not on taxonomies and terminological issues, these entity definitions are not presented here.
• “{[x]}” represents a pointer to an entity belonging to the class x.
• “{string}” represents a list of strings.
• “[{[x]}]” represents a list of pointers to entities belonging to the class x.
• “(I)” indicates that an attribute value and any constraints on that value are inherited from an abstract class. For example, an artifact has an attribute called name that is inherited from the abstract class DRP_Object. Because the name of any DRP_Object is required to be unique and not null, the name of any artifact is as well.
• “#” indicates that the rest of the line is a comment.
• “# (UNIQUE)” is a comment indicating that a string must have a unique value.
• “# (NOT NULL)” is a comment indicating that the field is required.

Abstract Class DRP_Object
{
  name string # (UNIQUE, NOT NULL)
  information [Information] # (NOT NULL)
  references [{[Reference]}]
  is_referenced_by [{[Reference]}]
  is_member_of [{[Set_Relationship]}]
  is_special_member_of [{[Directed_Set_Relationship]}]
}

Abstract Class Restricted_DRP_Object # (inherits from DRP_Object)
{
  name (I)
  information (I)
  constrained_by [{[Constraint]}]
  references (I)
  is_referenced_by (I)
  is_member_of (I)
  is_special_member_of (I)
  required_by [{[Requirement]}]
}
Class Artifact   # (inherits from DRP_Object)
{
    name (I)
    information (I)
    references (I)
    is_referenced_by (I)
    is_member_of (I)
    is_special_member_of (I)
    type [Artifact_Family] # (NOT NULL)
    is_specified_by {[Specification]}
    config_info [Config_Info] # (NOT NULL)
    function {[Function]} # (NOT NULL)
    form [Form] # (NOT NULL)
    behavior {[Behavior]}
    subartifacts {[Artifact]}
    subartifact_of {[Artifact]}
    is_source_of {[Flow]}
    is_destination_of {[Flow]}
}

Class Function   # (inherits from Restricted_DRP_Object)
{
    name (I)
    information (I)
    constrained_by (I)
    references (I)
    is_referenced_by (I)
    is_member_of (I)
    is_special_member_of (I)
    required_by (I)
    type [Function_Family] # (NOT NULL)
    subfunctions {[Function]}
    subfunction_of [Function]
    function_of_artifact [Artifact] # (NOT NULL)
}

Class Transfer_Function   # (inherits from Function)
{
    name (I)
    information (I)
    constrained_by (I)
    references (I)
    is_referenced_by (I)
    is_member_of (I)
    is_special_member_of (I)
    required_by (I)
    type (I)
    subfunctions (I)
    subfunction_of (I)
    function_of_artifact (I)
    input_flow {[Flow]}
    output_flow {[Flow]}
}
Class Flow  # (inherits from Restricted_DRP_Object)
{
  name (I)
  information (I)
  constrained_by (I)
  references (I)
  is_referenced_by (I)
  is_member_of (I)
  is_special_member_of (I)
  required_by (I)
  type [Flow_Family]  # (NOT NULL)
  source {{Artifact}}
  destination {{Artifact}}
  has_external_source Boolean  # (NOT NULL, Default FALSE)
  has_external_destination Boolean  # (NOT NULL, Default FALSE)
  is_input_of {{Transfer_Function}}
  is_output_of {{Transfer_Function}}
}

Class Form  # (inherits from Restricted_DRP_Object)
{
  name (I)
  information (I)
  constrained_by (I)
  references (I)
  is_referenced_by (I)
  is_member_of (I)
  is_special_member_of (I)
  required_by (I)
  type [Form_Family]  # (NOT NULL)
  subforms {{Form}}
  subform_of [Form]
  geometry [Geometry]  # (NOT NULL)
  material [Material]  # (NOT NULL)
  form_of_artifact [Artifact]  # (NOT NULL)
}

Class Geometry  # (inherits from Restricted_DRP_Object)
{
  name (I)
  information (I)
  constrained_by (I)
  references (I)
  is_referenced_by (I)
  is_member_of (I)
  is_special_member_of (I)
  required_by (I)
  type [Geometry_Family]  # (NOT NULL)
  subgeometries {{Geometry}}
  subgeometry_of [Geometry]
  geometry_of_form [Form]  # (NOT NULL)
}
Class Material # (inherits from Restricted_DRP_Object)
{
  name (I)
  information (I)
  constrained_by (I)
  references (I)
  is_referenced_by (I)
  is_member_of (I)
  is_special_member_of (I)
  required_by (I)
  type [Material_Family] # (NOT NULL)
  submaterials {[Material]}
  submaterial_of [Material]
  material_of_form [Form] # (NOT NULL)
}

Class Behavior # (inherits from DRP_Object)
{
  name (I)
  information (I)
  references (I)
  is_referenced_by (I)
  is_member_of (I)
  is_special_member_of (I)
  type [Behavior_Family] # (NOT NULL)
  subbehaviors {[Behavior]}
  subbehavior_of [Behavior]
  behavior_of_artifact [Artifact] # (NOT NULL)
}

Class Specification # (inherits from DRP_Object)
{
  name (I)
  information (I)
  references (I)
  is_referenced_by (I)
  is_member_of (I)
  is_special_member_of (I)
  type String # (NOT NULL)
  requirements {[Requirement]} # (NOT NULL)
  specification_of_artifact [Artifact] # (NOT NULL)
}

Class Config_Info
{
  name String # (UNIQUE, NOT NULL)
  information [Information] # (NOT NULL)
  type String # (NOT NULL)
  config_info_of_artifact [Artifact] # (NOT NULL)
}

Class Information
{
  name String # (UNIQUE, NOT NULL)
  description String
  documentation String
  methods (String)
  properties (String)
}
Abstract Class DRP_Relationship
{
  name String # (UNIQUE, NOT NULL)
  information [Information] # (NOT NULL)
}

Class Requirement   # (inherits from DRP_Relationship)
{
  name (I)
  information (I)
  type String # (NOT NULL)
  requires {
                  ([Restricted_DRP_Object]) # (NOT NULL)
  }
  requirement_of_spec {
                    ([Specification]) # (NOT NULL)
}
}

Class Reference   # (inherits from DRP_Relationship)
{
  name (I)
  information (I)
  type String # (NOT NULL)
  referred_object [DRP_Object] # (NOT NULL)
  referring_object [DRP_Object] # (NOT NULL)
}

Abstract Class Set_Relationship   # (inherits from DRP_Relationship)
{
  name (I)
  information (I)
  members {
                  ([DRP_Object]) # (No fewer than 2 DRP_Objects in list)
  }
}

Class Undirected_Set_Relationship   # (inherits from Set_Relationship)
{
  name (I)
  information (I)
  members (I)
  type String # (NOT NULL)
}

Class Directed_Set_Relationship   # (inherits from Set_Relationship)
{
  name (I)
  information (I)
  members (I)
  type String # (NOT NULL)
  special_members {
                    ([DRP_Object]) # (NOT NULL)
  }
  special_member_role String
  member_role String
}

Class Constraint   # (inherits from DRP_Relationship)
{
  name (I)
  information (I)
  type String # (NOT NULL)
  constrains {
                  ([Restricted_DRP_Object]) # (NOT NULL)
  }
}