An Object-Oriented Representation for Product and Design Processes

S. R. Gorti[*], A. Gupta[†], G. J. Kim[‡], R. D. Sriram[§], and A. Wong[¶]

December 22, 1997

[*]GeoQuest, Schlumberger, Houston, TX 77042.
[†]Dept. of Civil Engineering, Indian Institute of Technology, New Delhi, India.
[‡]Computer Science Department, POSTECH, Pohang, Korea.
[§]Engineering Design Technologies Group, NIST, Gaithersburg, MD 20899
[¶]Schlumberger Austin System Center, Austin, TX 78726.

Abstract

We report on the development of a knowledge representation model, which is based on the SHARED object model reported in [33, 34]. Our current model is implemented as a layered scheme, that incorporates both an evolving artifact and its associated design process. To represent artifacts as they evolve, we define objects recursively without a pre-defined granularity on this recursive decomposition. This eliminates the need for translations between levels of abstraction in the design process. The SHARED model extends traditional OOP in three ways: 1) by allowing explicit relationship classes with inheritance hierarchies, 2) by permitting constraints to be associated with objects and relationships, and 3) by comparing "similar" objects at three different levels (form, function and behavior). Five primitive objects define the design process: goal, plan, specification, decision and context. Goal objects achieve function, introduce constraints, introduce new artifacts or modify existing ones, and create subgoals. Plan objects order goals and link a product hierarchy to a process hierarchy. Specification objects define user inputs as constraints. Decision objects relate goals to user decisions and context objects describe the design context. Operators that are applied to design objects collectively form a representation of the design process for a given context. The representation is robust enough to effectively model four design paradigms (described in Reference [9]): top-down decomposition, step-wise refinement, bottom-up composition and constraint propagation. To demonstrate this, we represent the designs of two TV remote controllers in the SHARED architecture. The example reveals that certain aspects of artifact knowledge are essentially context-independent and that this representation can be a foundation for robust knowledge-based systems in design.

Key Words: Design, knowledge-base, object-oriented model, design process, ontology, representation

1 Introduction: The Comprehensive Design Knowledge-Base

Researchers have been exploring ways to represent design as a synthesis procedure for more than a quarter century [19], [21]. Early research on computer support for engineering design concentrated on problem-solving techniques (see Reference [27] for descriptions of various problem solving techniques). These techniques are relatively mature now, and design researchers realize that even good techniques operating on a weak representational research foundation must necessarily be inadequate to support engineering design. Consequently, recent trends have seen a shift in paradigm to an emphasis on representation issues; a recent issue (Volume 10, Number 4, September 1996) of the journal *Artificial Intelligence in Engineering Design, Analysis and Manufacturing* is dedicated to papers on representing function and behavior in design. The nature of engineering design and the diversity and complexity of engineering knowledge require knowledge representation schemas to be as flexible and robust as possible.

The search for a flexible and a robust knoweldge representation is inherently related to the view design researchers seek to model. Tong and Sriram [29] cite numerous definitions of design. One of which, by Tomiyama and Yoshikawa [28], is the "mapping of a point in function space onto a point in attribute space." Such a definition, and others like it, provides an indication that researchers focus on either representing design processes or the products these processes operate on. The

design research literature reflects this by exhibiting a dichotomy between process representation and product representation.

Process-based approaches usually model top-down functional decomposition of design, organizing the design product elements as functional primitives. In contrast, an alternate view holds design to be a bottom-up synthesis of component elements to constitute the whole, with the complex interplay and interconnections among these components serving to provide the overall system function. Kamal et al. [13] and Kim and Bekey [14] propose generic decision-based representations, while Ullman et al. [30] propose a mechanical design process model based on empirical data. Several papers in [29] describe other research in functional decomposition; additional references on design process modeling are provided by Braha and Maimon [3].

Product representation presents several additional research issues, and these have been well-documented. Object-oriented approaches to design have enabled a natural decomposition and hierarchical structuring of design product knowledge. The dynamically evolving nature of the composition hierarchies, evolving form descriptions, multiple functional and geometric abstractions and multiple levels of constraints have all been identified as crucial issues for product representation [34]. The representation must provide not only for the evolutionary nature of design process enaction, but also for the evolutionary nature of the domain description itself (as in development of comprehensive knowledge bases).

To represent generic routine design, Gero [8] developed conceptual schema that includes form, function and behavior within a single situation framework. Alberts et al. [2] extend this to innovative design by developing generic components that represent the bottom-up element of design. These are based on physical theory, and could represent combinations of basic components that implement commonly used behaviors. However, neither of these works incorporates the representation of an evolving design.

This paper addresses this very issue. More specifically, we address the issue of how to flexibly represent design knowledge in such a way that it supports layered development and step-wise refinement of comprehensive engineering knowledge bases. While our primary focus is on modeling the design products (artifacts), we believe that modeling the design process is at least as important for automating design processes and representating design histories. We present an integrated approach to modeling the design enterprise as a whole. This approach forms the basis for a conceptual design shell called CONGEN [9], which is a domain-independent knowledge-based design support framework, implemented as an application over a layered architecture in a modular, object-oriented manner. CONGEN supports design as a *synthesis* process, involving the arrangement of pre-defined "building blocks" to compose a design. The synthesis is based on an integration of four problem-solving approaches: process-based hierarchical decomposition (or alternately stated, top-down functional decomposition), step-wise refinement, product-oriented bottom-up models, and constraint propagation approaches. Details of these problem solving models are presented in References [9] and [27].

In developing comprehensive engineering knowledge bases, many different kinds of knowledge need to be represented. These include: knowledge about various objects, their properties, behavior, shape, and interrelationships among objects, as well as causal knowledge relating objects through physical phenomena, either quantitatively or qualitatively. This latter knowledge type sometimes

assumes the form of empirical observations and heuristics; this is what we often refer to as "experience." Whatever the form, most engineering knowledge is built upon layers of related knowledge, from the most basic physical principles through highly domain specific situational principles, including commonly used assemblies of physical systems. Our approach to developing comprehensive engineering knowledge bases is based on exploiting this layered structure. The comprehensive knowledge base represents standard design products at the domain level, and this representation draws upon descriptions of function, form and behavior from lower levels of the layered structure. The proposed representation scheme is geared to accommodate current and future ISO STEP-based standards [1].

The paper is organized as follows: Section 2 presents an object model, which forms a basis for our product-process representation. Based on this object model, we describe a representation scheme for design artifacts in Section 3. In Section 4, we describe a related scheme for representing design processes; we also present an integration of the two schemes, illustrated with an example. A discussion follows in Section 5, after which we present our summary and conclusions in Section 6.

2 An Object Model For Design Knowledge Representation

In this section, we present an object model that forms the basis for our design knowledge representation. Our model is based on the SHARED object model, which extends the object-oriented methodology [33] in three ways:

1. Instead of using only attribute references to objects, the model provides explicit relationship entities with associated semantics and constraints. These relationships are associated with relationship classes which can be arranged in inheritance hierarchies as with object classes.
2. The model associates constraints with objects and relationships. The knowledge representation scheme uses constraints to maintain the consistency and integrity of a product model.
3. The model provides a mechanism for comparing "similar" (i.e. interchangeable) objects in a meaningful way.

Below, we provide a brief overview of the SHARED object model.

2.1 Definition of Objects

A SHARED object, **o**, is defined as a *unique, identifiable entity* in the following form:

$$\mathbf{o} = (\mathbf{oid}, \mathbf{vid}, \mathbf{A}, \mathbf{M}, \mathbf{R}, \mathbf{C})$$

- **oid** is a unique identifier of an object (**o**); in this paper we will use various symbolic names for **oid**s instead of long computer generated numbers. The set of all unique object identifiers is denoted by **soid**.
- **vid** is a non-unique identifier similar to the object identifier. It is used to refer to a set of similar objects which can be used to replace one another in relationships. These similar objects have the same **vid**. Typically, we use this concept to model alternatives or versions.[1] Thes set of all **vid**s is **svid**.
- **A** is a set of three-tuples, (**t**, **a**, **v**). Each **a** is called an *attribute* of **o** and is represented by a *symbol* which is unique in **A**. Associated with each attribute is a *type*, **t** and a *value*, **v**. Each **t** has an associated domain, **domain(t)**, such that **v** $\in$ (**domain(t)** $\cup$ {**nil**}).
- **M** is a set of tuples, $(\mathbf{m}, \mathbf{tc_1}, \mathbf{tc_2}, ..., \mathbf{tc}_n, \mathbf{tc})$. Each element of **M** is a *method signature* which uniquely identifies a method. The symbol **m** represents a method name; methods define operations on objects. The symbols $\mathbf{tc}_i$, i = 1, ..., n, specify the argument type and **tc** specifies the returned value type.
- **R** is a set of relationships among **o** and other objects. Each relationship is identified by its unique identifier, **rid**. We discuss relationships again in Section 2.2.
- **C** is a set of constraints that are exerted on **o** or that exist between **o** and other objects. Each constraint within the set is defined by (**cname**, **code**), where **cname** is a unique identifier for the constraint, and **code** is its description, in an appropriate language. A constraint is often a boolean function that returns either *TRUE* or *FALSE*, indicating whether the constraint is met or not. Constraints may be used to restrict ranges of attributes; they can also be used to define complex expressions relating object attributes or rules which are to be satisfied. This component of the SHARED object model is an extension of other object-oriented representations.

As a simple, yet demonstrative example, consider the SHARED object shown in Table 1 which is formally specified as:

(**can-opener_1**, nil, {(real, weight, 7.2), (string, purpose,"open cans"), (char, material, steel)}, {(check_constraint(), ..., boolean), (assemble_part(),...,TRUE)}, {has_part_123}, {(c1, (< weight 10)), (c2, (= material (one-of steel aluminum)))}).

In this example, **can-opener_1** is a unique identifier, **real**, **char**, **string** are primitive data types, and **has_part** is a relationship. There is no **vid** because this is not a replaceable object. **c1** is an identifier for a constraint which specifies that the **weight** should be less than 10 newtons (N), while **c2** represents the constraint *the material is either steel or aluminum.*

[1] Alternatives or versions can be represented using relationships (e.g., by defining instances of alternative_of relationships among instantiated objects). Our view is that the concepts of alternatives and versions are used very frequently during design and deserve a more explicit representation and more direct access.

Table 1: A simple SHARED object.

Object oid can-opener_1			
instance_of		can-opener	derived from Artifact class
vid		nil	type int
A	purpose	"open cans"	type string
	weight	10	type real
	material	steel	type char[]
M		check_constraints()	constraint checker
		assemble_part()	..
R		has_part_123	part relationship with parents
C		(c1, (< weight 10))	constraint on weight
		(c2, (= material (one-of steel aluminum)))	constraint on material type

2.2 Relationships

The SHARED model explicitly represents relationships among objects. This aids in the design process described later. All objects have either an instance_of or a subclass_of (depicted by is_a) relationship. In the following discussions these two relationships are shown separately from **R**.

Typical types of relationships include ones that are compositional (such as **part_of/has_part**), functional (such as **satisfies/satisfied_by**, **has_subfunction/subfunction_of**), spatial (such as **connected_to**) and configurational (such as **has_version/version_of**). We define a generic SHARED relationship as follows:

$$\mathbf{r} = (\mathbf{rid}, \mathbf{RO}, \mathbf{A}, \mathbf{M}, \mathbf{C})$$

where

- **rid** is a unique identifier of the relationship **r**. The set of all unique relationship identifiers is denoted by **srid**.
- **RO** is a set of three-tuples, (**t**, **ro**, **v**).
  Each tuple (or element) of **RO** is called a *role* of the relationship. **ro** is the name of the role and **v** is the *value* of the role and **t** is the type of **v**; note that $\mathbf{v} \in \text{domain}(\mathbf{t})$, where domain(**t**) is some non-empty subset of subsets of $\mathbf{svid} \cup \mathbf{soid}$. As expected, there must be at least two objects partaking in the roles of a relationship. For a relationship between a particular set of objects to be valid, each of the objects must be identified by some role in the relationship and each of the objects must include the particular relationship in the relationship set **R** of the object's definition.
- **A** is a set of attributes of a relationship, defined in a manner similar to the set **A** of a SHARED object.

- **M** is a set of methods, defined in a manner similar to the set **M** of a SHARED object. The methods define operations on the roles and attributes of the relationships.
- **C** is a set of constraints on objects associated with the roles of the relationship and its attributes. It includes constraints on cardinality of roles. It is defined in the same way as the set **C** of a SHARED object.

A simple instance of **part_of/has_part** relationship is illustrated in Table 2 and in Figure 1, where **can-opener** is a composite part of type **System**. **Set_System** denotes another class (namely, a set of **System** objects), and **cutter**, **gear**, and **lever** are identifiers of objects which constitute this set (see Figure 1). This object represents a relation between a composite object (of type **System**) made of three subsystems (of an aggregate type of **Set_System**). The method `get_subsystem`, which has no arguments, is an access function that returns the subsystems.

Table 2: A simple SHARED relationship.

Relationship rid has_part_123			
instance_of		has_part	
RO	composite	can-opener	type System
	components	cutter, gear, lever	type Set_System
A	description	"can-opener has 3 subsystems"	type string
M		get_subsystem()	components selector
C		nil	

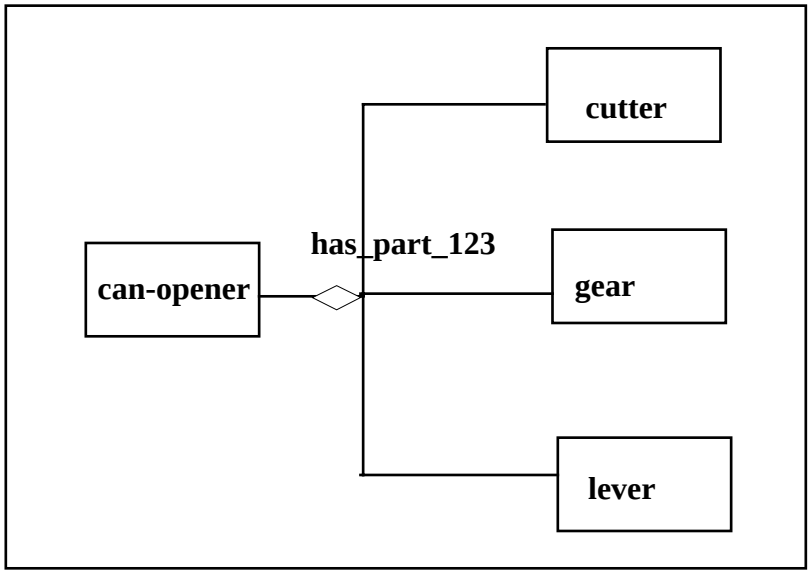

Figure 1: An instance of a has_part relationship object (notation used is from [20]).

2.3 Classes

Classes and **relationship classes** are defined on the objects and relationships defined above, as abstraction mechanisms to make common properties and semantics explicit. A SHARED object, **o**, is classified as an *instance* of a class, **c**, if **o** inherits all attributes, relationships, methods and constraints of the class **c**. For a more complete definition of a class in the SHARED object model see [33]. Similarly, a relationship **r**, is classified as an *instance* of a relationship class, **rc**, if **r** inherits all roles, attributes, methods and constraints of that relationship class, **rc**.

Generalization and specialization are also defined in terms of the class abstractions. These define a partial order on the set of all classes (i.e. they are *reflexive, antisymmetric*, and *transitive*). Generalization is also used as an implementation mechanism for sharing code and data types among more specialized classes. That is, a specialized class can inherit properties of a number of more general classes, in a process known as *multiple inheritance*.

This brief overview highlights the extensions to the traditional object-oriented model we made in SHARED; Wong and Sriram describe the model in detail in [34]. In the next two sections, we illustrate a representation scheme for design artifacts and their design processes based on the SHARED object model. Our example uses two TV remote controller designs, shown in photographs in Figure 2 and Figure 3.

3 Product Description

We define an **artifact** to essentially consist of **function, form**, and **behavior**. These three components of an **artifact** are represented explicitly as required attributes in the SHARED model with the usual notions of relationships, constraints, and methods (or rules).

An **artifact** as a SHARED tuple:

$$\mathbf{Artifact} = (\mathbf{oid}, \mathbf{vid}, \mathbf{A_{fun}} \cup \mathbf{A_{form}} \cup \mathbf{A_{beh}}, \mathbf{M}, \mathbf{R}, \mathbf{C})$$

where the attributes **A** of the shared object model now represent the triples: function $\mathbf{A}_{fun}$, form $\mathbf{A}_{form}$, and behavior $\mathbf{A}_{beh}$ of the artifact, respectively. Further, $\mathbf{A}_{fun}$ is a set of triples (**Function**, function, value), $\mathbf{A}_{form}$ is a singleton comprising the triple (**Form**, form, value), and $\mathbf{A}_{beh}$ is a set of triples (**Behavior**, behavior, value). Note that the value is a pointer to an object whose type will be a subclass of the appropriate type. For ease of notation we will use: $\mathbf{A}_{fun} = \{$(**Function**, function)$\}$, $\mathbf{A}_{form} =$ (**Form**, form), and $\mathbf{A}_{beh} = \{$(**Behavior**, behavior)$\}$.

As defined earlier, **M** is a set of methods, **R** is a set of the various relationships, and **C** is a set of constraints. Representative elements of **R** are: **introduces, modifies, has_part**. These and other relationships for encoding design rationale are described by Peña-Mora [17]. Figure 4 shows a hierarchy of classes for the remote control system example. Table 3 shows an example of two replaceable (i.e., same **vid**) artifact representations at their highest levels.

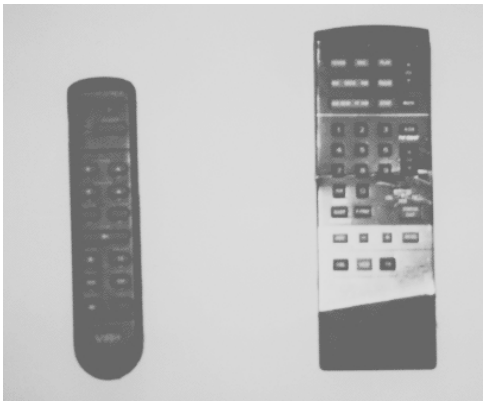

Figure 2: Two different television remote controls.

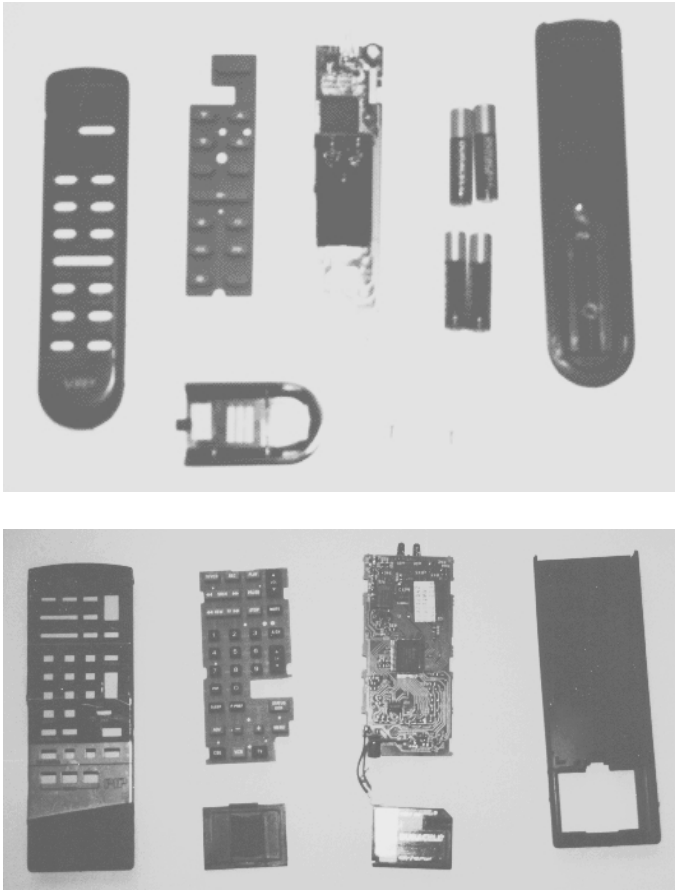

Figure 3: Components of the smaller and larger remote controls in Figure 2. Although the overall design structures are similar, there are many distinctions, including size, features, and mating types.

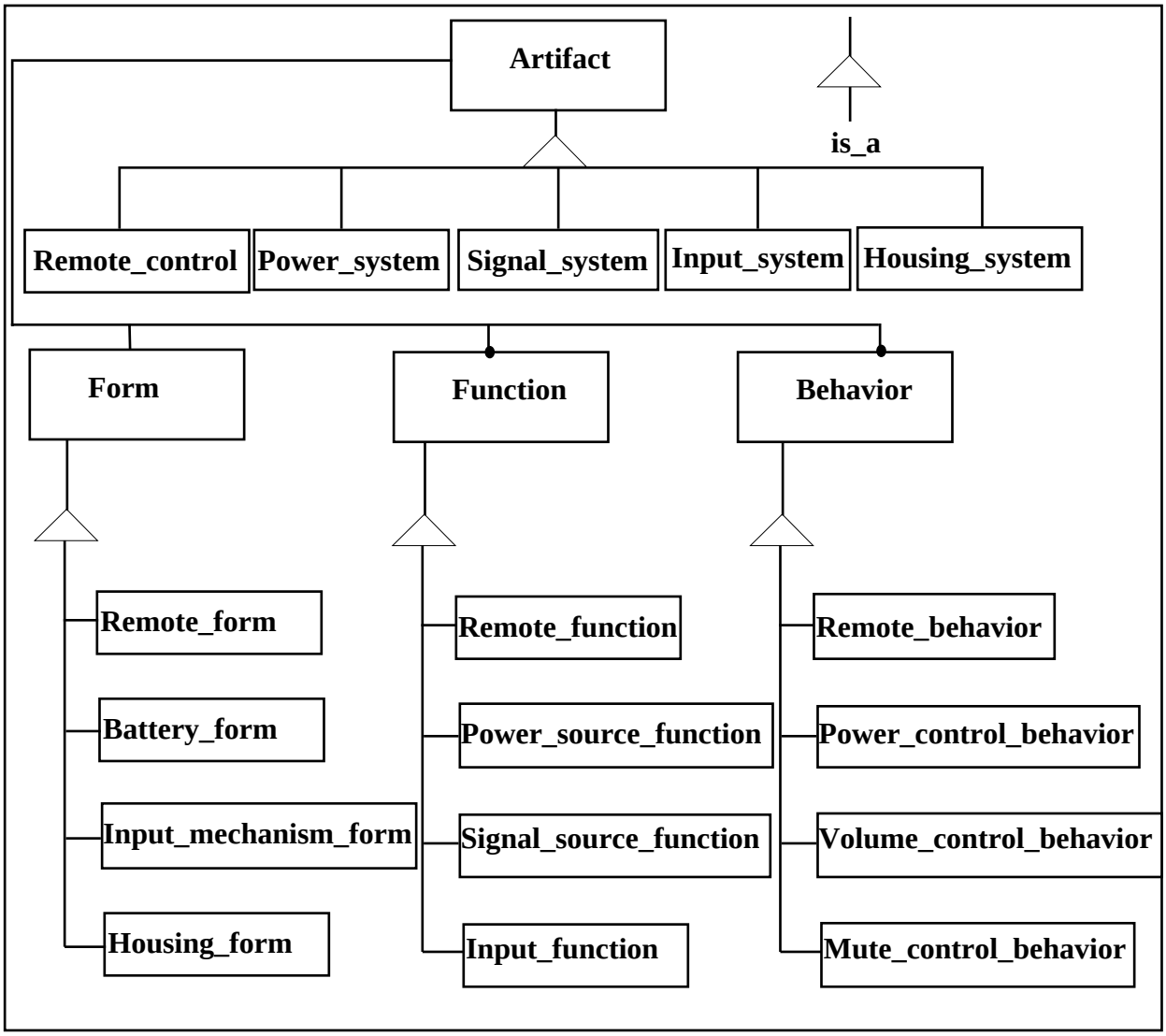

Figure 4: The class hierarchy (by is_a relationship) of the representation of the remote control system example in Tables 3-6. Only object classes are shown in this figure. Instances (not shown) are postfixed with a "_number," such as remote_form_1; Tables 3-6 display specific examples.

The **function**, **form**, and **behavior** objects are further described below.

1. **Function** objects, as exemplified in Table 4, are used to capture the intended functional requirements of a design problem. The various elements of our object model are as follows: **oid** and **vid** are identifiers; **A** is now {(**String**, description, value)} or simply {(**String**, description)}, description is a required textual description – this attribute can be alternatively expressed by pre-defined domain-specific functional vocabularies – of the **function** object, **M** is a set of methods (e.g., consistency checks on pre-conditions for the **function** to be satisfied) and **R** is a set of relationships. Typical relationship classes used among **function** objects and

Table 3: High level representation of one remote control system system, remote_control_1. The other remote control, remote_control_2, has a similar representation; the instance_of, vid and the methods are the same, but the relationship and the attributes are different. For example, in place of a name such as remote_form_1, there is remote_form_2. The artifact is represented by three aspects: function, form, and behavior, which are also objects. The relationship, **has_part_1**, identifies four subsystems of the **remote_control_1**, and the constraint, **interface_spec_1**, places a restriction on choices of certain subsystems, in this case, a keypad system as a user input mechanism. Constraints, relations and other attributes of the objects are generated and changed dynamically during the design process.

Object oid remote_control_1			
instance_of		Remote_control	derived from Artifact class
vid		vid_56	belongs to group of replaceable objects
A	Function	remote_function_1	
	Form	remote_form_1	
	Behavior	remote_behavior_1	
M		make_part()	method for instantiation
		get_subsystem()	method for retrieving its subsystems
		check_constraint()	general method for checking constraints
R		has_part_1	must specify subsystems
C		(interface_spec_1 ...) (...)	this remote control model should have keypad as its input system

Relationship rid has_part_1			
instance_of		Has_part	
RO	composite	remote_control_1	
	components	power_system_1, signal_system_1, input_system_1, housing_system_1	
	description	"remote has 4 subsystems"	

other types of objects (e.g., behavior, form) include:

achieves/achieved_by (between **function** and **behavior**),
satisfies/satisfied_by (between **function** and **artifact**),
requires/required_by, and
has_subfunction/subfunction_of (among **function** objects).

A **function** object must have either a **satisfies/satisfied_by** relationship with another object or have a **has_subfunction/subfunction_of** relationship to other lower level functions. **C** is a set of constraints.

2. **Form** objects represent physical properties such as structure, geometric shape, and material. The various terms of our object model have the following connotations: **oid** and **vid** are identifiers, **A** is now {(**Space**, space, value), (**Property**, property, value)} or simply {(**Space**, space), (**Property**, property)}. (**Space**, space) represents spatial information such as envelope volume, position, orientation, and spatial relationships and (**Property**, prop-

Table 4: Lower level functional objects for representing the remote control system. The **function** object (**remote_function_1**) is broken down further into four subfunctions forming an hierarchy (by **subfunc_1 relation**). One of the subfunctions, **power_source_function_1**, is **satisfied_by** an **artifact** object, called **power_system_1**.

Object oid remote_function_1			
instance_of		Remote_function	derived from Function class
vid		vid_2	belongs to group of replaceable Function objects
A	description	"remotely control television functions: power, volume and channel"	
R		has_subfunction_1	must specify subsystems
Relationship rid has_subfunction_1			
instance_of		Has_subfunction	
RO	function	remote_function_1	
	subfunction	power_source_function_1, signal_source_function_1, input_function_1, housing_function_1	
	description	"remote has 4 subfunctions"	
Object oid power_source_function_1			
instance_of		Power_source_function	derived from Function class
vid		vid_9	
A	description	"provide 6 Volt of electric power"	
R		has_subfunction_1, satisfied_by_1	
Relationship rid satisfied_by_1			
instance_of		Satisfied_by	
RO	function	power_source_function_1	
	artifact	power_system_1	
A	description	"power source is satisfied by a separate subsystem"	

erty) represents a set of non-geometric attributes such as material requirements, assembly hierarchy, and surface finish. **M** is a set of access methods including queries about spatial relationships and physical properties, geometric transformations, and display selection operators. This set can include an accumulation method, which accumulates properties of objects related through composition relationships, and also includes a top-level method for propagating operations through relationships. **R** is a set of relationships, and **C** is a set of spatial consistency constraints (e.g., 3D abstraction should enclose lower level abstractions). Table 5 presents an example of the form object for the remote control devices.

3. **Behavior** objects specify the response of an artifact to input conditions or behavioral states (or both). Using a predefined set of domain-dependent vocabularies that describes various actions and behavioral states, we represent behavior with input and output relationships among processes, states and actions. Iwasaki and Chandrasekaran [12] and Vescovi et al [32] present similar specifications for engineering behavior. The various terms of our object model have the

Table 5: Two lower level structural objects for representing the remote control system. The **form** object (**remote_form_1**) consists of four components. The constraint, `material_spec_1`, specifies that one of the components, **input_mechanism_form_1**, be made of rubber. This constraint looks for a particular type of form (in this case, that of class **Input_mechanism_form**) and tests if its attribute `material` has a value **rubber**.

Object oid remote_form_1				
instance_of			Remote_form	derived from Form class
vid			vid_67	group of replaceable Form objects
A	space	3D_geometry	remote_control_1.sld	3D solid CAD file
		position	(100, 50, 200)	
		orientation	(0, 0, 0)	
		size	(1, 1, 1)	
	property	material	nil	this is an assembly
		component	power_system_form_1, signal_source_form_1, input_mechanism_form_1, housing_form_1	
M			get_bounding_box(), get_total_weight()	
C			(material_spec_1, (equal (get_value (find_form parent.component Input_mechanism_form)) material))	input mechanism must be made of rubber

Object oid input_mechanism_form_1				
instance_of			Input_mechanism_form	derived from Form class
vid			vid_45	group of replaceable Form objects
A	space	3D_geometry	keypad.sld	3D solid CAD file
		position	(80, 50, 100)	
		orientation	(0, 0, 0)	
		size	(1, 1, 1)	
	property	material	rubber	
M			get_bounding_box()	

following connotations: **oid**, **vid** are identifiers; **A** is now {(**Input**, input, value), (**Output**, output, value), (**String**, description, value)} or simply {(**Input**, input), (**Output**, output); (**String**, description)}; input represents input actions and conditions; output represents the resulting behavioral states and actions in response to the input conditions, description is a textual description of the behavior of the object; **M** is a set of methods for computing the output behaviors of components or deriving required input conditions of a given output state; **R** is a set of relationships with other objects (e.g., **function** and **form**) for which this behavior is binding; and **C** is a set of behavioral constraints. See Table 6 for an object description of remote control behavior.

Each of these objects may be a *composite* object, at various levels of abstraction relevant to the granularity of the system description, e.g., **behavior** objects may be represented at various levels

by qualitative descriptions, by approximate models or by exact equations. Similarly, **space** objects represent the many levels of geometric abstraction that may be relevant to a description of the physical existence of an artifact: 3D, wire-frame, symbolic (engineering idealizations), and sectional views, for example. In general, the constraints and lower dimensional abstractions must be enclosed within the 3D geometry abstraction [33]. The examples in Tables 3-6 illustrate a partial view of the overall object model of the remote control system.

So far, we have presented an object-oriented representation scheme for an artifact, and illustrated it with snapshots of the artifact in three respects (i.e., form, function, and behavior); the representation is implementated in a commercial object-oriented database management system and is described by Murdock et al. [16]. Now we turn to representing the design process.

4 Representing Design Processes

Describing a design process requires knowledge about artifact synthesis. Activities often associated with design processes include setting and achieving goals, satisfying specifications, making decisions and following a plan of action. Based on these generic activities, and others, we created a set of objects to capture design process. The following sections describe these process objects and design operators.

4.1 Process Objects

We now describe primitive objects relevant to representing a process. Like SHARED objects for representating products, process objects have five basic components: **oid**, **vid**, type-attribute-value triplets (**A**), a set of methods (**M**), a set of relationships (**R**), and a set of constraints (**C**). Beyond these basic components, process objects contain five entities: **Goal**, **Plan**, **Specification**, **Decision**, **Context**. Each is described below.

(a) As indicated in Figure 5, **Goal** objects act as central decision points for tasks in the design process, and are the links between product and process models. A **goal** may achieve a function in the functional hierarchy, and the function may serve as a reference into the product world descriptions in the domain knowledge (product) database. More generally, a **goal** introduces a constraint, modifies an artifact, introduces a new artifact, or creates further subgoals. Alternately, we may pursue the *subgoals*, which constitute moves within the process hierarchy. These moves are further dictated by two considerations: testing of alternatives specified for the **goal**, and selections made by the user from a set of valid alternatives.

We represent **goals** this way to have a set of methods that relates to retrieving and matching procedures (discussed in Section 4.2). A **goal** has relationships that associate it to a parent **plan** in the process hierarchy and also to other entities, as shown in Figure 5. A **goal** is also related to the artifact by relationship classes, **creates** and **modifies**. Constraints control the interactions among conflicting **goal** objects. Fromont and Sriram, and Gorti et al. [7, 11] describe similar constraint processing algorithms.

Table 6: Objects for representing remote control system behavior. The **behavior** objects (**remote_behavior_1**) and (**remote_behavior_2**) are each composed of one relationship, and one method, which computes the overal **behavior** object from its component subbehaviors. One alternative, **has_subbehavior_1**, has three relationship roles, or subbehaviors, associated with it (power control, volume control and channel control), while the other alternative ha four (an additional mute control signal). One of the subbehaviors, **volume_control_behavior_1**, specifies the input and output relations using predefined and domain dependent vocabularies (e.g., in the input event of *press_1*, the resulting state is a "volume" signal.)

Object oid remote_behavior_1			
instance_of		Remote_behavior	derived from Behavior class
vid		vid_89	group of replaceable behavior objects
A	description	"emit signals: power, volume, channel"	
M		calc_composite_behavior()	calculate behavior (not implemented)
R		has_subbehavior_1	
Object oid remote_behavior_2			
instance_of		Remote_behavior	derived from Behavior class
vid		vid_89	group of replaceable behavior objects
A	description	"emit signals: power, volume, channel and mute"	
M		calc_composite_behavior()	calculate behavior
R		has_subbehavior_2	
Relationship rid has_subbehavior_1			
instance_of		Has_subbehavior	
RO	behavior	remote_behavior_1	
	subbehavior	power_control_behavior_1, volume_control_behavior_1, channel_control_behavior_1	
A	description	"3 subbehaviors"	
Relationship rid has_subbehavior_2			
instance_of		Has_subbehavior	
RO	behavior	remote_behavior_2	
	subbehavior	power_control_behavior_1, volume_control_behavior_1, mute_control_behavior_1, channel_control_behavior_1	
A	description	"4 subbehaviors"	
Object oid volume_control_behavior_1			
instance_of		Remote_behavior	derived from Behavior class
vid		vid_17	group of replaceable behavior objects
A	input	press_1	input action (predefined vocabulary)
	output	{ volume_up_signal volume_down_signal }	output states
	description	"emit signals which means volume up or down"	
R		has_subbehavior_1	

(b) **Plan** objects prioritize **goal** objects in a meaningful way. **Plan** objects are also associated with artifacts and link the product into the process hierarchy; the reverse link, from the process to the product is through **goal** objects. A **plan** may be associated with a set of planning rules, which set priorities on the subgoals.

The methods of a **plan** object set the schedule for task achievement as an ordering on the **goal** objects. We define planning rule bases to achieve this ordering based on the current design *context* conditions. **Plan** objects have relationships that associate themselves to their parent **goal** objects or **artifact** objects in the process hierarchy, depending on whether the move being pursued is a refinement of the function or a product decomposition.

(c) **Specification** objects represent user inputs that involve important bottom-up elements of the design process. They are formally specified and defined as constraints on relationships and attribute values.

(d) **Decision** objects refer to all user decisions that govern choices for further expansion of a **goal**. Therefore, they have relationships to the relevant **goal** objects and attributes for providing textual rationale information. A **goal**, as we stated above, represents a decision point, and the **decision** objects record the alternatives chosen for a **goal** within a given design context. Each new **decision** spurs a new design context (potentially an entirely different design alternative), and we allow the designer to pursue *multiple* design alternatives simultaneously. **Decision** objects further capture the justifications for validity of each alternative as generated by the system, and also the rationale for the choice by the user.

(e) **Context** describes the design context that represents a particular design alternative. A **context** thus consists of the design tasks (**goal** objects) relevant to the current design alternative, the user specifications (or constraints), the decisions that have been made, and relevant product information. **Contexts** aid in managing multiple design alternatives.

These entities are related as shown in the class abstractions in Figure 5. Further details of these classes can be found in [9]. The product-process interactions are shown as links between the process objects and design objects.

4.2 Design Operators (Methods)

4.2.1 Design Process

As shown in Figure 6, we view a design process as a sequence of mappings from one design state to another until one or more acceptable artifacts are found; the design operators responsible for the mappings are applied to design objects.

The design flow proceeds from describing a function to describing an artifact, and includes selecting a form for the product. However, we require neither the functional hierarchy to be fully pre-specified nor the product hierarchy to be pre-specified. This implies that the further decomposition of a chosen artifact may well determine the further *functional* or *product hierarchy. Form* determination

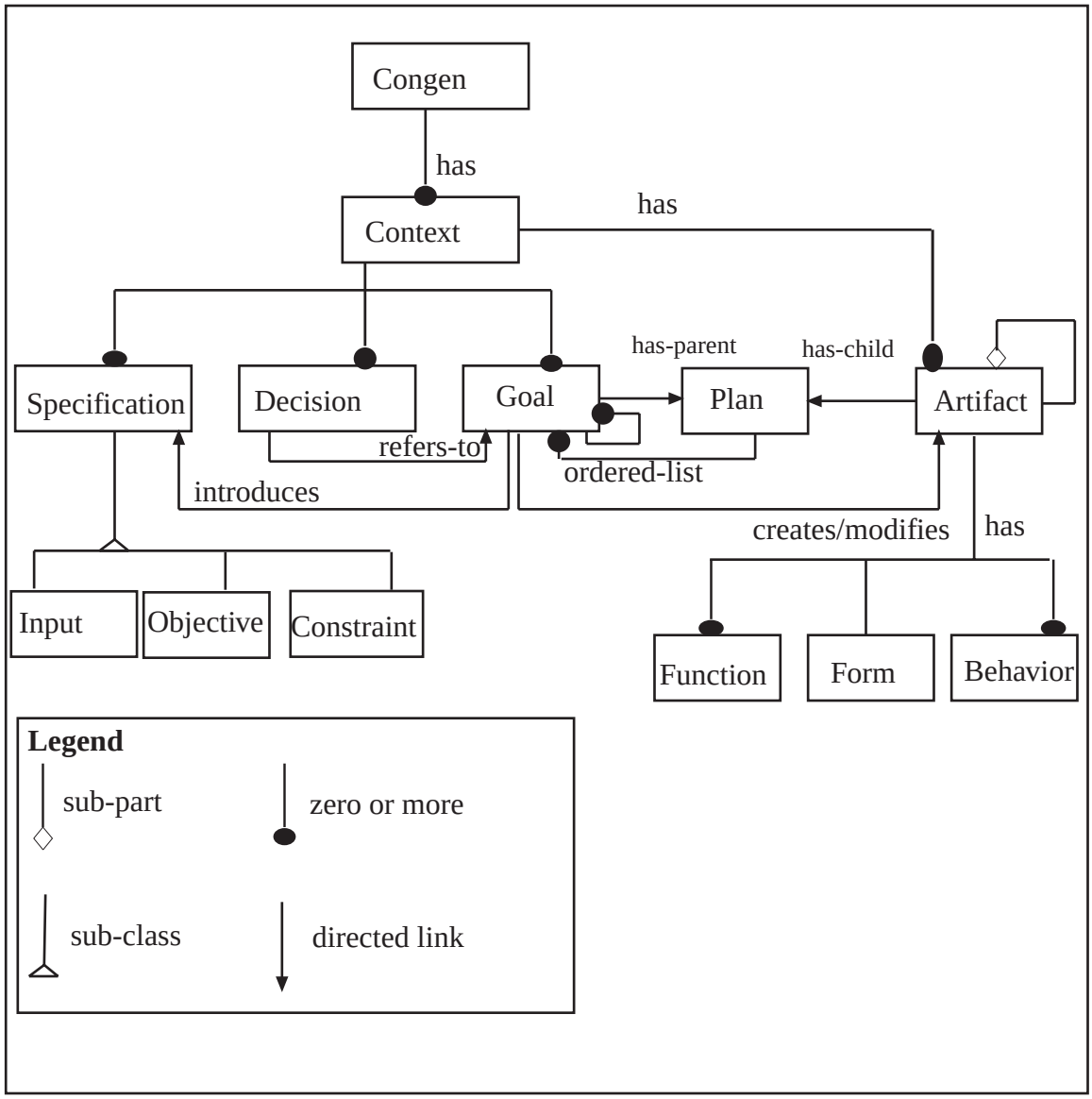

Figure 5: CONGEN class abstractions (from[9] with permission).

at any stage includes specifying the topological connectivity of the components and their structural relationships, while details of geometry of the components are relegated to the next stage. *Behavior* cannot be examined in isolation of the structural configurations. But determining *form* allows us to analyze *behavior*, and to examine the feasibility of the *form* chosen to satisfy a given *function*.

4.2.2 Goal Execution

There are two categories of methods by which a goal is satisfied. These are: making design changes or further expanding a **goal**:

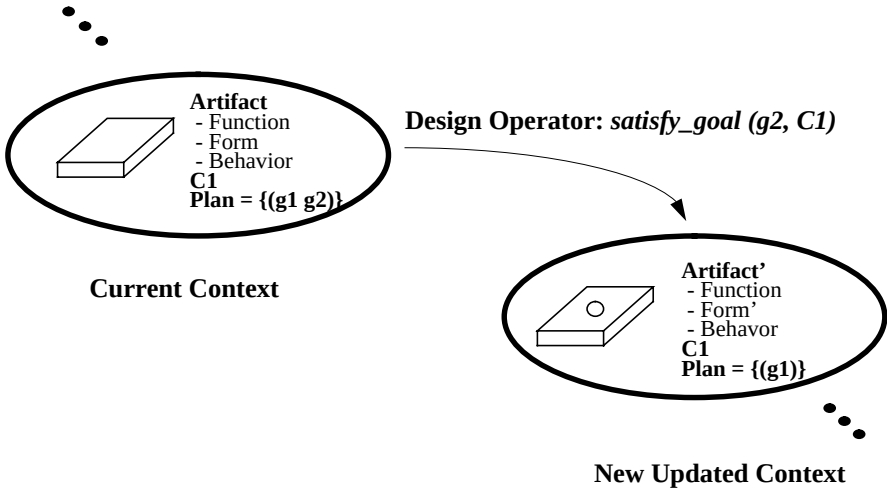

Figure 6: A model of one design process. By satisfying design goals and carrying out design plans, design operators update design contexts. In this figure, a goal, **g2**, is selected from the current plan, **Plan**, and a design operator is applied to satisfy **g2**, that results in a new form, called **Form'** and an updated context.

$$\begin{aligned} \textit{satisfy\_goal}(\mathbf{G}, \mathbf{CT}) \rightarrow \quad & (\textit{modify}(\mathbf{G}, \mathbf{CT}) \mid \textit{expand\_goal}(\mathbf{G}, \mathbf{CT})) \ \& \ \textit{verify\_constraints}(\mathbf{Set\_CT}_I) \\ \rightarrow \quad & \mathbf{Set\_CT}_{new} \end{aligned}$$

where **G** is the selected goal for the current context, **CT**, $\mathbf{Set\_CT}_I$ is a set of new interim contexts and $\mathbf{Set\_CT}_{new}$ is the final set of new contexts.

This operator represents the pursuit of a design task that may lead to a set of potential alternatives. In both cases, a set of new contexts, representing new design alternatives, is generated.

Even when a given **goal** is satisfied, it is still subject to a verification for constraint violations. In general, modifying an **artifact**, or expanding into further subgoals entails a set of newly generated constraints. To complete a goal satisfaction operator, the `verify_constraints` operator must return a TRUE value for the newly generated context.

Modifying the current context can occur by one of the following: (1) changing the attribute values of artifact-related objects (such as **form**, **function**, and **behavior**); (2) establishing a new relationship or deleting an existing relationship among existing artifact-related objects; or (3) deleting existing objects, creating new design objects, or retrieving design objects from a database. Form, function, or behavior snippets can be used for retrieval, as described in Reference [27], Chapter 6. Context changes can be made either automatically or by user intervention.

$$
\begin{aligned}
\mathit{modify}(\mathbf{G}, \mathbf{CT}) \rightarrow \quad & \mathit{change\_attribute\_value}(\mathbf{G}, \mathbf{CT}) \quad | \\
& \mathit{establish\_relationship}(\mathbf{G}, \mathbf{CT}) \quad | \\
& \mathit{delete\_relationship}(\mathbf{G}, \mathbf{CT}) \quad | \\
& \mathit{delete\_object}(\mathbf{G}, \mathbf{CT}) \quad | \\
& \mathit{create\_object}(\mathbf{G}, \mathbf{CT}) \quad | \\
& \mathit{find\_object}(\mathbf{G}, \mathbf{CT})
\end{aligned}
$$

where **G** is the selected goal of a given context, **CT**. The `expand_goal` operator generates new subgoals, and attaches them to the existing plan for reordering.

$$
\mathit{expand\_goal}(\mathbf{G}, \mathbf{CT}) \rightarrow \mathbf{Set\_G}_{new} \;\; \& \;\; \mathit{update\_plan}(\mathbf{Set\_G}_{new}, \mathbf{P}_{new}) \;\; \& \;\; \mathit{order\_plan}(\mathbf{P}_{new})
$$

where $\mathbf{Set\_G}_{new}$ is a set of new subgoals and $\mathbf{P}_{new}$ is the newly formed plan.

4.2.3 Design Decision

Attempting to satisfy a design goal may result in multiple design contexts for a user to pursue. To reduce this number, a design decision must be made to pursue or abandon a particular design thread. The following operators represent this notion.

$$
\begin{aligned}
\mathit{pursue\_context}(\mathbf{CT}_{selected}) \rightarrow \quad & \mathbf{CT}_{current} \\
\mathit{pursue\_goal}(\mathbf{G}_{selected}) \rightarrow \quad & \mathbf{G}_{current}
\end{aligned}
$$

where $\mathbf{CT}_{selected}$, $\mathbf{CT}_{current}$, $\mathbf{G}_{selected}$, $\mathbf{G}_{current}$ each represents the user-selected and current contexts and goals, respectively.

Figure 7 illustrates an evolution and generation of design contexts (left side) and a description of its process (right side). Initially, the context starts with an artifact, **A**=remote, and an initial top-level goal, $\mathbf{G}_{initial}$= "design remote control." To satisfy $\mathbf{G}_{initial}$, a `modify` operator executes an `expand_goal` operator, which expands the initial goal into two alternative plans (i.e., **G1** then **G2**, or **G2** then **G1**). Also, the constraints (not shown) are verified. The user in this example chooses to pursue **G2**, the goal to find the power system. To satisfy the goal, a `find_object` operator locates two possible components for the **power_system**, namely, a 9V battery or two AA batteries. This creates a new design context (or alternative), **C2**, which is pursued in both context and process, and a decision is made to pursue **C2** and a rationale for justifying this decision is created. Figures 8 and 9 show two possible conceptual geometric solutions to the power-system problem.

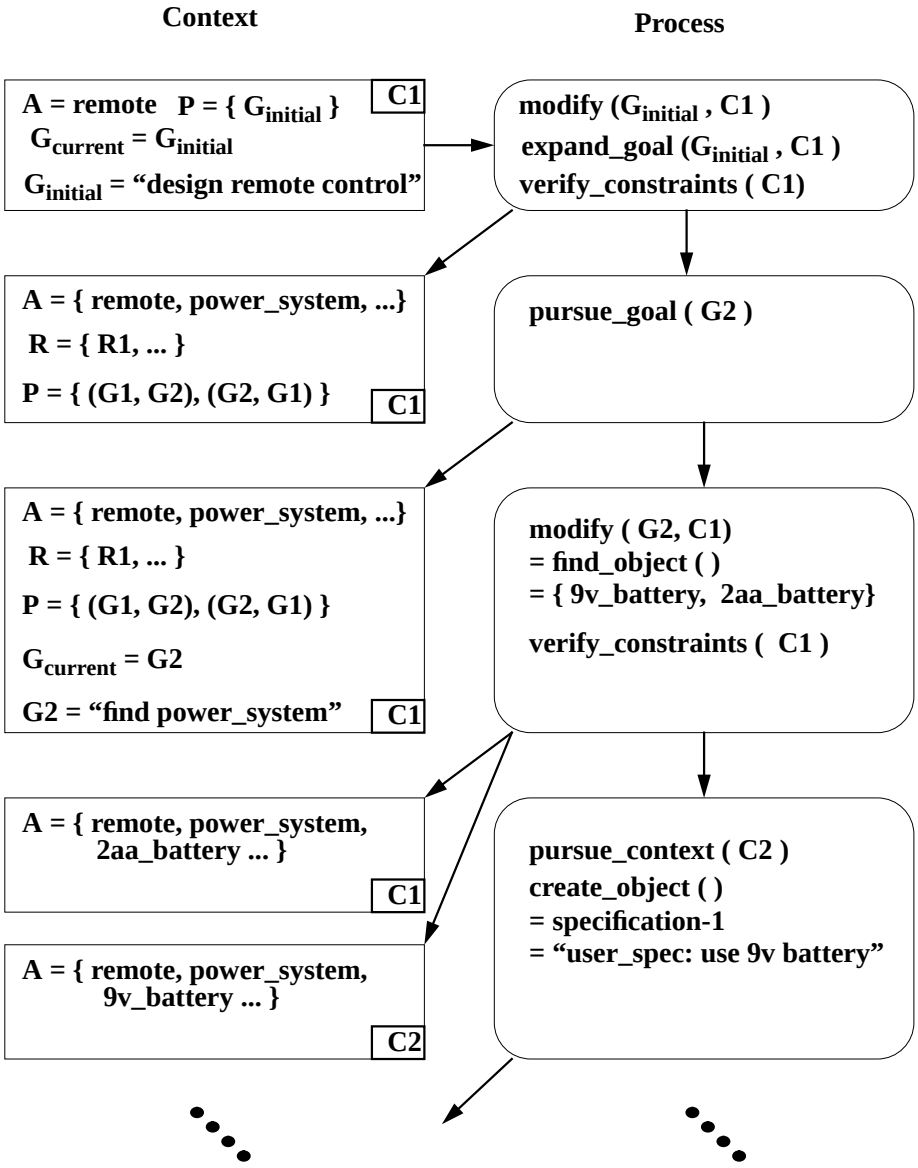

Figure 7: Representation of the remote control system artifact and its associated design process.

5 Discussion

A product is the physical result of a practical design process, a mapping from an abstract functional description to tangible entities. *Function* often implies physical *behavior*, and decisions on sub-systems directly impact the mechanisms chosen for achieving the *behavior* to realize a given *function*. That the *form* relationships between the sub-systems affect the *behavior* of the overall system further complicates the problem. We thus realize that a robust, flexible representation must necessarily view *function*, *form* and *behavior* from an integrated viewpoint. We make an important conceptual distinction between representation of structure and structural relationships versus behavior and behavioral relationships. However the behavior of a device cannot be examined in isolation to its

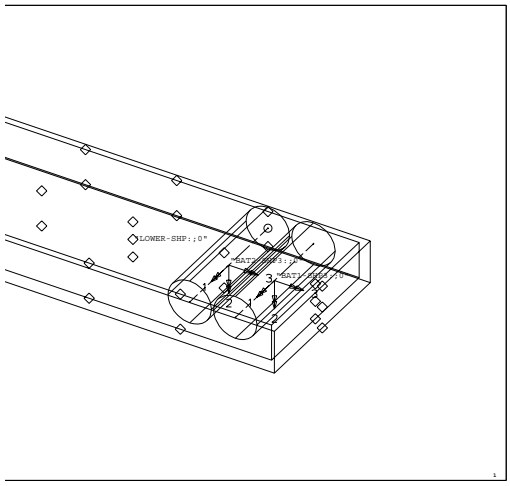

Figure 8: A possible battery configuration in context, C1 (a conceptual geometric model).

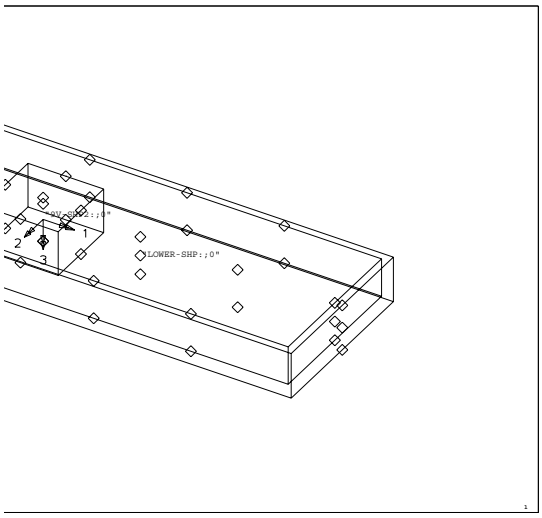

Figure 9: A possible battery configuration in context, C2 (a conceptual geometric model).

structural configuration, and the causal patterns encode this knowledge. The behavioral description of a device states only *what the device does*, not *how it is achieved*, and this is consistent with recent object-oriented design approaches. Causality is linked directly to the components of the device, and this is an important conclusion with ramifications for design synthesis.

In our case, function is dependent on context, while behavior is dependent on form relationships, and as such is context-independent. Context encompasses *viewpoint* in this case, and contains design-specific information. We speak of function being view-dependent, and make a case for a separation of context-dependent and context-independent information. In other words, an object can have multiple behaviors independent of the context as its inherent property. However, certain of these behaviors are normally instantiated in a particular context, manifested in one or more functions. We exploit this clear separation of context-dependent and context-independent representational elements to support the goal of comprehensive knowledge bases. Thus the context-independent knowledge is comprised of **artifacts** (akin to the concepts represented by *generic components* in [2]) whereas the context information provides the knowledge required to combine these concepts in functionally useful ways.

This allows making the components truly *generic*, as argued by Alberts et al. [2]. Yet, the representation of function is a compromise that we make for allowing evolution of the comprehensive knowledge bases. While we argue that function representation is legitimately a part of the design process, and as such function-form mapping is generally through a set of expected behaviors, *routine* design incorporates the notion of a direct function-form mapping. If each artifact description introduced into the knowledge base were to encapsulate the functions, routine design would thus be facilitated without an extensive re-haul of the existing process knowledge. This compromise comes from a realization that a representation tailored to be flexible must still not be cumbersome from a routine design standpoint. Additional rationalization for the above arguments is provided in Reference [10].

The generic components proposed by Alberts et al. are similar to our own definition of artifacts, and are complementary to the design prototypes of Gero [8] in a manner similar to our design processes and products being complementary. Our approach differs in allowing an evolving representation of an artifact, obviating the need for translations between so-called "technology-based" layers, which represent levels of abstraction in the design process [2]. Our artifacts are recursively defined to consist of further artifacts, and we place no pre-defined granularity limits on this recursive decomposition. Related work on ontologies and design representation can be found at the following web sites: http://www.ksl.stanford.edu/; and http://www.ie.utoronto.ca/EIL/eil.html

6 Summary and Conclusions

In our representation model, we capture two separate aspects of design: artifact description (where the basic constructs to model the problem are defined), and subsequent design process enaction (which operates on these constructs to lend coherence and overall structure to the design). The process enaction stage must follow from the model description stage, and to a large extent, flexibility afforded by the model description governs the innovation in the process. Our approach to model description aims to encapsulate the physical principles inherently involved in engineering design,

through a structured approach to knowledge encoding. This approach forces the structure of design knowledge at various levels, from knowledge of physical principles common to engineering problems through domain-specific heuristic knowledge. The comprehensive knowledge bases thus capture "deep" knowledge about the design products in the form of physical behavior, and provide the basis for knowledge-based design support systems. The underlying theme in each one of these descriptions is the motivation to support development of comprehensive engineering knowledge bases.

In our layered representation, design operators update and transform design contexts, and support a range of automation, extending from purely manual design (in which the designer uses knowledge in an explicitly modeled process) all the way to automated design (with reasoning mechanisms). Our current implementation uses rules and constraints, coupled with human intervention, in decision making. The design concepts are separated into context-dependent and context-independent parts, representing the bottom-up and top-down knowledge respectively. Artifact knowledge is identified as being essentially context-independent. The granularity of the artifact descriptions governs, in part, the innovativeness of the design. The model formulation and behavior verification phases allow for physical principles to drive the process of retrieval. The representation schema thus forms the basis for a second-generation knowledge based design support framework. We have not yet completely addressed the problems of schema integration invariably associated with a common comprehensive product model for a collaborative enterprise: research on this issue is currently being pursued.

7 Acknowledgments

We gratefully acknowledge members of the DICE group at MIT, in particular Feniosky Peña-Mora, for many useful discussions with us. Gerard Kim was supported by the National Research Council Postdoctoral Research Associateship. Ashok Gupta was a visiting scientist from the Dept. of Civil Eng., IIT, Delhi to MIT, and was supported by a Indo-US Science and Technology Fellowship from the Dept. of Science and Technology, Govt. of India and the United States Agency for International Development. Funding for the DICE project came from the IESL affiliates program and a NSF PYI Award No. DDM-8957464, with matching grants from NTT Data, Japan and Digital Equipment Corporation, USA. We thank Robert Allen and J. William Murdock for pointing out several errors and rectifying these errors in the manuscript. We would also like to thank Y. Narahari for his comments on the final draft of the paper. The object and the design process models were developed by Sriram and his students at MIT, and later refined by Gerry Kim at NIST. The authors' names are in alphabetical order.

References

[1] STEP: The Future of Product Data Exchange, Automotive Industry Action Group, 26200 Lahser Road, Suite 200, Southfield, MI 48034 (More information can be found at http://www.nist.gov/sc4 and http://www.nist.gov/sc5/soap/).

[2] Alberts, L, K., Wognum, P.M., and Mars, N. J. I., "Structuring Design Knowledge on the Basis of Generic Components," *AI in Design*, Gero, J. S. (Editor), Kluwer Academic Publishers, 1992.

[3] Braha, D. and Maimon, O., "The Design Process: Properties, Paradigms, and Structure," IEEE Transactions on Systems, Man, and Cybernetics, Vol. 27, No. 2, pages 146–166, March 1997.

[4] Bylander, T. and Chandrasekaran, B. "Understanding Behavior using Consolidation," Proceedings of IJCAI-85, pages 23-34, Morgan Kaufman Publishers, 1985.

[5] Carey, M.J., DeWitt, D.J., Graefe, G., Haight, D.M., Richardson, J.E., Schuh, D.T., Shekita, E.J., and Vandenberg, S.L. "The EXODUS Extensible DBMS Project: An Overview," *Readings in Object-Oriented Databases,* S. Zdonik and D. Maier (Editors), Morgan-Kaufman, 1990.

[6] Davis, R., "Diagnostic Reasoning Based on Structure and Behavior," In *Qualitative Reasoning about Physical Systems*, Bobrow, D. (editor), MIT Press, pages 347–410, 1985.

[7] Fromont, B. and Sriram, D., "Constraint Satisfaction as a Planning Problem," in *AI in Design*, Gero, J. (Editor), pages 97–117, Kluwer Academic Press, Norwell, MA.

[8] Gero, J.S. "Design Prototypes: a Knowledge Representation Schema for Design," *AI Magazine,* pages 26-36, Winter, 1990.

[9] Gorti, S. and Sriram, R. D. "Symbol to Form Mapping: A Framework for Conceptual Design,", *Journal of CAD*, Vol. 28, No. 11, pages 853–870, 1996.

[10] Gorti, S. and Sriram, R. D. "From Symbol to Form: A Framework for Design Evolution,", Technical Report Number IESL 94-02, Department of Civil Engineering, MIT, Cambridge, MA 02139, 1994.

[11] Gorti, S., Humair, S., Sriram, R. D., Talukdar, S. and Murthy, S. "Solving Constraint Satisfaction Problems Using A-Teams," *Artificial Intelligence in Engineering Design And Manufacturing,* Vol. 10, No. 1, pages 1–20, 1996.

[12] Iwasaki, Y., and Chandrasekaran, B., "Design Verification through Function and Behavior-Oriented Representations: Bridging the Gap between Function and Behavior," in *AI in Design,* J. S. Gero (Editor), Kluwer Academic Publishers, 1992.

[13] Kamal, S., Karandikar, H., Mistree, F. and Muster, D., "Knowledge Representation for Discipline-Independent Decision Making," in *Expert Systems in Computer Aided Design,* Gero, J. (Editor), pp. 289-321, Elsevier, 1987.

[14] Kim, G. J. and Bekey, G., "Design-for-Assembly by Reverse Engineering," in *Artificial Intelligence in Design 94*, Gero, J. S. (Editor), pages 717-734, Kluwer Academic Publishers, 1994.

[15] Mittal, S., Dym, C., and Morjaria, M., "PRIDE: An Expert System for the Design of Paper Handling Systems," *IEEE Computer,* pages 102-114, July, 1986.

[16] Murdock, J. W., Szykman, S., and Sriram, R. D., "An Information Modeling Framework to Support Design Databases and Repositories," *Proceedings of the ASME 1997 Design Engineering Technical Conferences*, Paper No. DETC97/DFM-4373, Sacramento, CA, September, 1997.

[17] Peña-Mora, F., *Design Rationale for Computer Supported Conflict Mitigation during the Design-Construction Process of Large-Scale Civil Engineering Systems*, Sc. D. thesis, Department of Civil and Environmental Engineering, MIT, Cambridge, MA 20319, September 1994.

[18] Peña-Mora, F., Sriram, D., and Logcher, R., "SHARED-DRIMS: SHARED Design Recommendation-Intent Management System," in *2nd IEEE Workshop on Enabling Technologies Infrastructure for Collaborative Enterprises (WET ICE),* IEEE Press, 1993.

[19] Rodenacker, W., *Methodisches Konstruieren,* Springer, New York, 1971.

[20] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modeling and Design,* Prentice Hall, 1991.

[21] Simon, H., "The Sciences of the Artificial," Second Edition, MIT Press, 1981.

[22] Sriram, R. D., Cheong, K., and Kumar, L., "Engineering Design Cycle: A Case Study and Implications for CAE," in *Knowledge Aided Design,* M. Green (Editor), pages 117–156, Academic Press, 1992.

[23] Sriram, R. D., Logcher, R., Groleau. N., and Cherneff, J., "DICE: An Object-Oriented Programming Environment for Cooperative Engineering Design," in *AI in Engineering Design,* Vol. III, Tong, C. and Sriram, D. (Editors), pages 303–366, Academic Press, 1992

[24] Sriram, R. D., Wong, A., and He, L., "GNOMES: An Object-Oriented Non-manifold Geometric Engine," *Journal of CAD,* Vol 27, No. 11, pages 853–868, 1995.

[25] Sriram, D., Stephanopoulos, G., Logcher, R., Gossard, D., Groleau, N., Serrano, D., and Navinchandra, D., "Knowledge-Based Expert Systems in Engineering Design: Research at MIT," *AI Magazine*, pages 79–96, Fall, 1989.

[26] Sriram, R. D., et al., *An Object-Oriented Knowledge Based Building Tool for Engineering Applications,* IESL Research Report R91-16, Intelligent Engineering Systems Laboratory, M.I.T, 1991.

[27] Sriram, R. D., *Intelligent Systems for Engineering: A Knowledge-based Approach*, Springer Verlag, 1997.

[28] Tomiyama, T. and Yoshikawa, H., "Extended General Design Theory," in *Design Theory for CAD, Proceedings of the IFIP WG5.2 Working Conference 1985, Tokyo*, Yoshikawa, H. and Warman, E.A (Editors), pages 95-130, Elsevier, North-Holland, Amsterdam, 1986.

[29] Tong, C. and Sriram, D. "Introduction," *Artificial Intelligence in Engineering Design,"* Tong, C. and Sriram, D. (Editors), pages 1–53, Academic Press, 1991.

[30] Ullman, D., Dietterich, T. and Stauffer, L., "A Model of the Mechanical Design Process based on Empirical Data," *AI EDAM,* Vol. 2, pages 33-52, 1988.

[31] Umeda, Y., Takeda, H., Tomiyama, T., and Yoshikawa, H., "Function, Behavior and Structure," in *Applications of Artificial Intelligence in Engineering V,* Gero, J. S. (Editor), pages 177-193, Computational Mechanics Publications/Springer Verlag, 1990 (see also their recent work in AIEDAM, September 1996).

[32] Vescovi, M., Iwasaki, Y., Fikes, R., and Chandrasekaran, B., "CFRL: A Language for Specifying the Causal Functionality of Engineered Devices," *Proceedings of the Eleventh National Conference on Artificial Intelligence,* pages 626–633, AAAI/MIT Press, 1993.

[33] Wong, A. and Sriram, D., *Shared Workspaces for Computer-Aided Collaborative Engineering,* Technical Report, IESL 93-06, Intelligent Engineering Systems Laboratory, Department of Civil Engineering, MIT, March, 1993.

[34] Wong, A. and Sriram, D., "SHARED: An Information Model for Cooperative Product Development," *Research in Engineering Design,* pages 21–39, Fall, 1993.