Interactive Constraint-Aided Conceptual Design

Barry O'Sullivan
Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland.
b.osullivan@cs.ucc.ie

Abstract

Engineering conceptual design can be defined as that phase of the product development process during which the designer takes a specification for a product to be designed and generates many broad solutions to it. This paper presents an constraint-based approach to supporting interactive conceptual design. The approach is based upon an expressive and general technique for modeling: the design knowledge which a designer can exploit during a design project; the life-cycle environment which the final product faces; the design specification which defines the set of requirements that the product must satisfy; and the structure of the various schemes that are developed by the designer. A computational reasoning environment based on constraint filtering is proposed as the basis of an interactive design support tool. Using such a tool, human designers can be assisted in interactively developing and evaluating a set of schemes which satisfy the various constraints that are imposed on the design.

Keywords: Constraint processing, conceptual engineering design, interactive design.

1 Introduction

This paper is concerned with the development of a constraint-based approach to supporting interactive engineering conceptual design. Engineering conceptual design can be regarded as that phase of the engineering design process during which the designer takes a specification for a product to be designed and generates many broad solutions to it. Each of these broad solutions is generally referred to as a *scheme* [19]. Each scheme should be sufficiently detailed that the means of performing each function in the design has been fixed, as have any critical spatial and structural properties of, and relationships between, the principal components.

It is generally accepted that conceptual design is one of the most critical phases of the product development process. It has been reported that more than 75% of a product's total cost is dictated by decisions made during the conceptual phase of design [27]. Furthermore, poor conceptual design can never be compensated for by good detailed design [28].

In supporting interactive conceptual design a number of issues must be considered:

- The conceptual design process is initiated with a statement describing the desired properties of the required product. This statement may not be complete or it may be modified during design;
- Conceptual design is a process during which synthesis of a scheme is a fundamental activity;
- The human designer should have the freedom to approach the process in any way she wishes;
- In so far as it is possible, human designers should be alerted to any inconsistencies which exist in their designs;
- Designers may seek explanations for inconsistencies in their schemes or justifications for why certain options are available to them;
- Designers may wish to have explained to them how a particular scheme has come about;

- Automated evaluation and comparison of multiple schemes throughout the design process is necessary to focus the designer on promising alternatives.

It was these considerations that motivated and set the agenda for the research reported here.

This paper presents an interactive constraint-based approach to supporting a human designer during engineering conceptual design. The approach is based upon an expressive and general technique for modeling: the design knowledge which a designer can exploit during a design project; the life-cycle environment which the final product faces; the design specification which defines the set of requirements that the product must satisfy; and the structure of the various schemes that are developed by the designer. A computational reasoning environment based on constraint filtering [6, 7] is proposed as the basis of an interactive conceptual design support tool. Using such a tool, the designer can be assisted in developing and evaluating a set of schemes which satisfy the various constraints that are imposed on the design. In particular, the designer can be assisted in synthesizing a number of alternative schemes for the required product. The consistency of each scheme is constantly monitored, as is the consistency of each scheme with respect to the design specification and the other schemes which have been developed. Explanations and justifications can be generated to aid the designer's understanding of the state of the design problem using a known approach from the literature [6]. Arbitrary constraints can be asserted or retracted by the designer which permits the incorporation of new requirements into the design specification and give the designer freedom to approach the process as she wishes.

The remainder of the paper is organized as follows. Section 2 presents an overview of the literature on constraint processing for design. Section 3 presents a brief overview of the theory of conceptual design upon which the research presented in this paper is based. Section 4 discusses how this theory can be modeled in a constraint programming language. Section 5 presents a detailed example of interactive conceptual design based on constraint filtering. Section 6 compares the approach presented here with the most relevant literature. In Section 7 a number of concluding remarks are made.

2 A Review of Constraint Processing for Design

Most decisions that are made in daily life involve considering some form of restriction on the choices that are available. For example, the destination to which someone travels has a direct impact on their choice of transport and route: some destinations may only be accessible by air, while others can be reached using any mode of transport. Formulating decision problems in terms of parameters and the restrictions that exist between them is an intuitive approach to modeling them. These general restrictions can be referred to as "constraints".

The fact that constraints are ubiquitous in many decision problems has given rise to the emergence of many popular problem-solving paradigms based on the notion of constraints. These techniques have been widely reported in the literature in such research fields as Operations Research (OR) and Artificial Intelligence (AI).

Some of the most popular approaches to solving problems comprising a set of constraints defined on a set of parameters stem from the constraint processing paradigm. Constraint processing is concerned with the development of techniques for solving the *Constraint Satisfaction Problem* (CSP) [32]. A large number of problems in AI, computer science, engineering and business can be formulated as CSPs. For example, many problems related to machine vision, scheduling, temporal reasoning, graph theory, design, design of experiments and financial portfolio management can be naturally modeled as CSPs.

In the engineering design literature three phases of design are generally identified: conceptual design, embodiment design and detailed design [38]. During conceptual design the designer searches for a set of broad solutions to a design problem, each of which satisfies the fundamental requirements for the desired product. The embodiment phase of design is traditionally regarded as the phase during which an initial physical design is developed. This initial physical design requires the determination of component arrangements, initial forms and other part characteristics [50]. The detailed phase of design is traditionally regarded as the phase during which the final physical design is developed.

Constraint-based applications for design have been more commonly applied to the post-conceptual phases of design. These later phases of design are concerned with developing a subset of the schemes generated during the conceptual design phase into fully detailed designs. The CADET system uses simulated annealing as the basis for its constraint satisfaction algorithm for solving the constraint-based representation of the geometric product model [49, 51]. The IDIOM system uses constraint solving on geometric parameters for floor-planning [31], *Space*Solver uses the notion of solution spaces, defined by sets of constraints on continuous domains, as a basis for supporting interactive design [30]. A constraint-based knowledge compiler for parametric design in the mechanical engineering domain called MECHANICOT has been proposed [34]. Many constraint-based systems reported in the literature have been developed for supporting reasoning about purely geometric aspects of design for use with CAD systems [4, 20, 21, 47]. The use of constraint processing techniques for supporting configuration design has also been widely reported in the literature [16, 18, 33, 44, 45].

Modern approaches to product development, such as Concurrent Engineering [5], Integrated Product Development [2] and Design Coordination [15], attempt to maximize the degree to which design activities are performed in parallel. A number of researchers in the constraint processing community have developed constraint-based technologies that support integrated approaches to product development [7, 37]. One of the critical issues that must be addressed in supporting integrated design is the issue of conflict resolution and negotiation. Constraint-based approaches to managing conflict in collaborative design systems have been reported [3, 30, 26]. Using constraints to co-ordinate distributed agents in engineering design has also been reported [40, 41].

Constraint-based approaches to supporting conceptual design have been reported in the literature for quite a number of years [25, 46]. However, most of this research does not address the synthesis problem; the vast majority has focused on constraint propagation and consistency management relating to more numerical design decisions. For example, "Concept Modeler" is based on a set of graph processing algorithms that use bipartite matching and strong component identification for solving systems of equations [46]. The Concept Modeler system allows the designer to construct models of a product using iconic abstractions of machine elements. However, a number of issues are not addressed by this work. Amongst these issues is the dynamic nature of conceptual design. During conceptual design constraints may be added or deleted at any point. In addition, the system does not address the issue of design synthesis, nor does it address the comparison of alternative solutions to a design problem. However, Concept Modeler demonstrated that constraint processing did offer a useful basis for supporting designers working through particular aspects of the conceptual design problem.

Based on the earlier work on Concept Modeler, a system called "Design Sheet" has been developed [10, 43]. This system is essentially an environment for facilitating flexible trade-off studies during conceptual design. It integrates constraint management techniques, symbolic mathematics and robust equation solving capabilities with a flexible environment for developing models and specifying tradeoff studies. The Design Sheet system permits a designer to build a model of a design by entering a set of algebraic constraints. The designer can then use Design Sheet to change the set of independent variables in the algebraic model and perform trade-off studies, optimization and sensitivity analysis.

Some researchers have used the Dynamic Constraint Satisfaction Problem as a basis for managing conflict during the preliminary phases of engineering design [22, 26]. Traditional conflict resolution techniques in constraint-based models of the design process use *backtracking* and *constraint relaxation.* Some researchers focus on differentiating between types of assumptions that are made by designers during design. Variations on this type of approach have also been proposed for managing conflict in collaborative design [3].

The use of autonomous agents to solve CSPs for conceptual design has been reported in the literature [24]. The motivation for the work is the support of spatial layout generation. The constraint specification used in the work facilitates a high-level representation and manipulation of qualitative geometric information. The search engine used in the proposed system is based on a genetic algorithm. The issue of constraint consistency is not addressed in the work. In addition, important design issues such as synthesis are not considered. However, it is realized that the primary focus of this work is the use of autonomous agents to solve CSPs.

The use of constraint logic programming for supporting reasoning about dynamic physical systems has been reported [17]. This work combines a constraint logic programming approach with bond graphs to assist in the development of a simulation model of a system in the form of a set of differential equations. The approach can be used for identifying causal problems of a bond graph model of a dynamic physical system.

In Section 1 a number of issues were highlighted as being critical to effectively supporting interactive conceptual design. While all of the systems which have been reviewed here support some of these issues, few have attempted to address them all. The research presented here is an attempt to develop an approach which can support each of these issues in the context of interactive conceptual design. In Section 3 the design theory upon which our approach is built will be presented.

3 A Theory of Conceptual Design

The model of conceptual design adopted in this research is based on the hypothesis that during the design process a designer works from an informal statement of the requirements that the product must satisfy and generates alternative configurations of parts which satisfy these requirements. Central to this exercise is an understanding of function and how it can be provided. Figure 1 illustrates this model of conceptual design. While the model is based on a well-known approach to conceptual design [38], its implementation is novel.

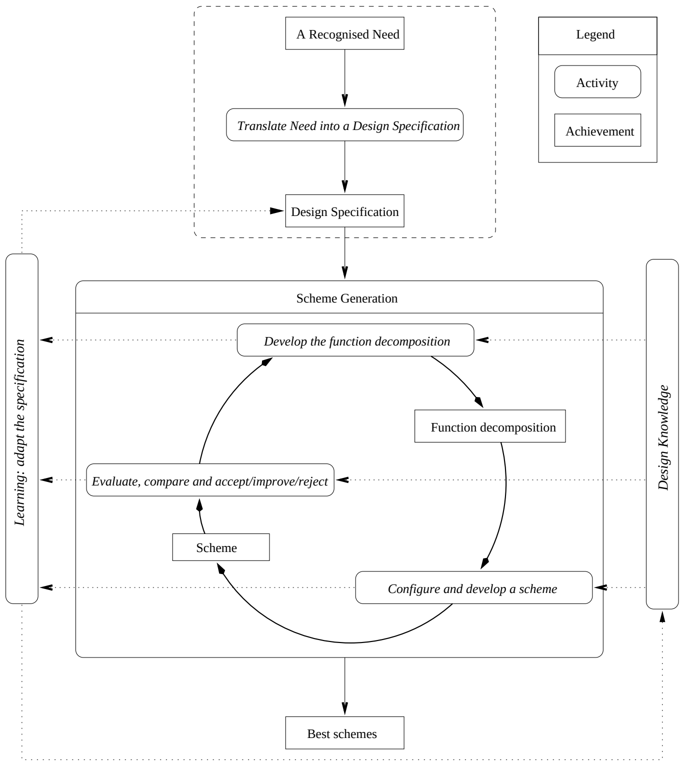

Figure 1: The model of the conceptual design process adopted in this research.

From Figure 1 it can be seen that the conceptual design process can be regarded as a series of activities and achievements. These activities and achievements relate to the development of the design specification and an iterative process of scheme generation. The process of scheme generation involves the development of a function decomposition which provides the basis for a configuration of parts that form a scheme. This scheme is then evaluated and compared against any schemes that have already been developed. Based on this comparison, the designer will choose to accept, improve or reject particular schemes. The process of scheme generation will be repeated many times in order to ensure that a sufficiently large number of schemes have been considered. The role of design knowledge and learning are also illustrated in Figure 1, using dotted lines. Design knowledge is used during the process of scheme generation. During scheme generation the designer may develop a greater understanding of the design problem being addressed; this learning may affect the design specification for the product or the design knowledge used to generate schemes.

In the remainder of this section a brief overview of the theory of conceptual design used in this research will be presented. For a more complete discussion of the theory the reader is encouraged to refer to the more detailed literature available [35].

3.1 The Design Specification

The conceptual design process is initiated by the recognition of a need or customer requirement. This need is analyzed and translated into a statement which defines the function that the product should provide (referred to as a functional requirement) and the physical requirements that the product must satisfy. This statement is known as a *design specification.*

Two categories of design requirement can be identified: *functional* requirements and *physical* requirements. A design specification will always contain a single functional requirement, since this represents the highest level of abstraction of a product; it may also contain a set of physical requirements which define the tangible characteristics of the required product.

In addition, two classes of physical requirement can be identified: product requirements and life-cycle requirements. A product requirement can be either a categorical requirement that defines a relationship between attributes of the product or it can be a preference related to some subset of these attributes. A life-cycle requirement can be either a categorical requirement that defines a relationship between attributes of the product and its life-cycle, or it can be a preference related to some subset of these attributes.

3.2 Conceptual Design Knowledge

During conceptual design, the designer must synthesize a configuration of parts which satisfies each of the functional and physical requirements in the design specification. To do so, the designer needs considerable knowledge of how function can be provided by physical means. Often, this knowledge exists in a variety of forms; a designer may not only know of particular components and technologies that can provide particular functionality, but may be aware of abstract concepts which could also be used. For example, a designer may know that an electric light-bulb can generate heat or, alternatively, that heat can be generated by rubbing two surfaces together. The latter concept is more abstract that the former. In order to effectively support the human designer during conceptual design, these alternative types of design knowledge need to be defined and modeled in a formal way.

3.2.1 The Function-Means Map

The notion of the *function-means tree* has been proposed by researchers from the design science community as an approach to cataloging how function can be provided by means [1]. The use of function-means trees in supporting conceptual design has attracted considerable attention from a number of researchers [9]. In general, the level of interest in the use of functional representations in conceptual design has increased in recent times [13], showing growing confidence in the potential of approaches incorporating such techniques.

In the research being described here a generalization of the function-means tree called a *function-means map* is used to model functional design knowledge [35]. A function-means map can be used to reconcile functions with means for providing them. In a function-means map two different types of means can be identified: a means can either be a *design principle* or a *design entity.*

A design principle is a means which is defined in terms of functions that must be embodied in a design in order to provide some higher-level functionality. The functions that are required by a particular design principle collectively replace the function being embodied by the principle. These functions will, generally, have a number of *context relations* defined between them. These context relations describe how the parts in the scheme which provide these functions should be configured so that the design principle is used in a valid way. An example design principle based on the abstraction of a bicycle is show in Figure 2. In this figure, the functions required by the design principle are illustrated using round-edged boxes while the context relations are illustrated using dashed boxes with dashed lines between the functions to which they apply.

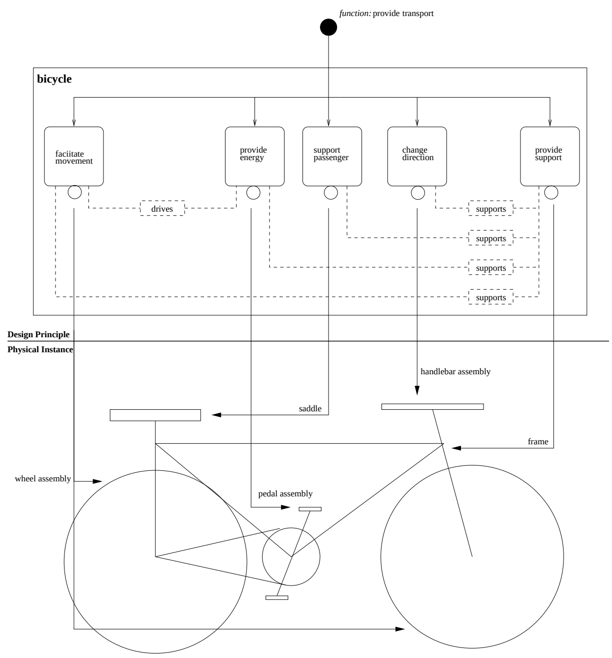

Figure 2: Abstracting an example design principle from a bicycle.

A design entity, on the other hand, is a physical, tangible means for providing function. A design entity is defined by a set of parameters and the relationships that exist between these parameters. For example, an electronic resistor would be modeled as a design entity which is defined by three parameters, resistance, voltage and current, between which Ohm's Law would hold.

3.2.2 Embodiment of Function

As the designer develops a scheme every function in the scheme is embodied by a means. Each means that is available to the designer has an associated *set of behaviours*. Each *behaviour* is defined as a set of functions that the means can be used to provide simultaneously. Each behaviour associated with a design principle will contain only one function to reflect the fact that it is used to decompose a single function. However, a behaviour associated with a design entity may contain many functions to reflect the fact the there are many combinations of functions that the entity can provide at the same time. For example, a design entity based on an electric light bulb may be able to fulfill the functions *provide light* and *generate heat* simultaneously. However, when a design entity is incorporated into a scheme (for the purpose of supporting functionality provided by one of its behaviours), it is not necessary that every function in this behaviour be used in the scheme.

Knowing the various possible behaviours of an entity is useful when the designer is embodying functions. Knowledge of behaviour enables a designer to identify when a particular design entity can be used to provide several functions simultaneously. In this research, the mapping of several functions onto a single design entity is known as *entity sharing*. Entity sharing is a critical issue in design because, without the ability to reason about entity sharing, a designer has no way of removing redundant parts from a design.

From a design perspective one of the novel aspects of the work presented here is the manner in which design principles and design entities are defined and used. Using design principles to define abstract design concepts in terms of functions and relationships between them is novel. The ability to represent means at both an abstract functional level as well as a physical level provides a basis for a designer to combine and explore new approaches to providing the functionality defined in the design specification.

3.3 Scheme Configuration using Interfaces

When a designer begins to develop a scheme for a product, the process is initiated by the need to provide some functionality. The designer begins to develop a scheme to provide this functionality by considering the various means that she has available in her design knowledge-base. Generally, the first means that a designer will select will be a design principle. This design principle will substitute the required (parent) functionality with a set of child functions.

As the designer develops a scheme and produces a function decomposition tree she will ultimately embody all leaf-node functions in the scheme with design entities. During this embodiment process the context relations, from the design principles used in the scheme, will be used as a basis for defining the interfaces between the design entities used in the scheme. However, the precise nature of these interfaces cannot be known with certainty until the designer embodies functions with design entities.

The types of interfaces that may be used to configure a collection of design entities will be specific to the engineering domain within which the designer is working. For example, the set of interfaces that a designer working in the mechanical domain would typically use are different to those used by a designer in the electrical domain. Indeed, these interfaces may also be specific to the particular company to which the designer belongs.

3.4 Scheme Generation

From Figure 1 it can be seen that scheme generation is an iterative process comprising a number of activities. These activities relate to the development of a function decomposition for the scheme, the development of a configuration of design entities based on this function decomposition and an evaluation and comparison of the newly developed scheme with those schemes which have already been developed. As new schemes are developed, the designer will constantly consider which schemes to accept, reject or improve.

Once the design specification has been formulated the designer must attempt to develop as many schemes as possible which have the potential to satisfy the requirements in the design specification. There are generally many function decompositions possible from the same functional requirement. Generating alternative function decompositions can be regarded as a way of systematically exploring the design space for a product. This search is controlled by a designer who considers the functional requirement for the product and explores alternative ways of providing the functionality by using design principles to decompose the functional requirement into less abstract functions. In this way the designer can translate the functional requirement into one which allows the use of standard technologies, represented as design entities, to satisfy it.

Using a function decomposition the designer can begin to develop a configuration of design entities. Each leaf-node function in a particular function decomposition must be provided by a design entity. The context relations inherited from the branch-nodes in the function decomposition, due to the use of particular design principles, are used to define the manner in which the design entities used in the scheme should be configured or interfaced. Some context relations will define constraints on the spatial relationships between design entities, while others may define particular types of interface which may be required between the entities. As the designer incorporates design entities into a scheme the various physical requirements which are described in the design specification can be brought to bear on it. Thus, the process of scheme generation presented here bears some similarities to some systematic approaches to configuration [48]. However, the similarities with configuration are discussed elsewhere [35, 36].

3.5 Evaluation and Comparison of Schemes

The designer's primary objective during conceptual design is to develop a set of alternative schemes which satisfy both the functional and physical requirements defined in the design specification. This set of schemes will be further refined during subsequent stages of the product development process until a small number of fully specified designs, possibly just one, will be selected for commercialization.

Every scheme must satisfy the categorical product and life-cycle requirements that are defined in the design specification. If a scheme violates one of these the designer must either reject the scheme which has been developed or modify it in order to satisfy them.

However, not every one of these schemes will be selected for development in subsequent phases of design. This is due to particular schemes not "satisfying" the preferences that were defined in the design specification. In this research the principle of Pareto optimality is used to determine which schemes are best from those developed by the designer [35]. Each design preference can be regarded as an objective function. Thus, the best schemes that are developed are those that are not dominated, in the Pareto optimal sense, by any other scheme. Obviously, in order to compare two schemes they must be based on the same design specification.

Table 1: Using the principle of Pareto optimality to compare two schemes.

Property	*Preference*	*First Scheme*	*Second Scheme*
number of parts	minimal	6	5
mass	minimal	$< x$	x

For example, Table 1 presents a comparison of two schemes based on two preferences: that the number of parts in the design be minimal and that the mass of the artifact be minimal. It can be seen that the first scheme comprised six parts while the second scheme comprised five. Thus, on the preference for a scheme comprising a minimal number of parts, the second scheme is better than the first. At this point it is not possible to determine if the first scheme is dominated. However, if the first scheme is not to be dominated by the second scheme it will have to have a smaller mass than the second scheme. In this way, each scheme would be better than the other on one design preference.

Using the principle of Pareto optimality provides a useful basis for comparing alternative schemes. It can be used to identify schemes that are completely dominated by schemes which have already been developed; this should motivate a designer to modify a scheme so that it is no longer dominated. Using the principle of Pareto optimality a designer can compare schemes that have been developed in order to select those that will be developed further and in order to identify those schemes that should be either improved or discarded. However, a designer should not be *forced* to discard dominated schemes. The objective is to motivate the designer to consider ways of improving them.

4 A Constraint-Based Implementation for Interactive Design

In this section an approach to developing the back-end of an interactive design tool for supporting conceptual design will be presented. By back-end we refer to the underlying reasoning engine upon which a graphical user-interface could be built. The reasoning system, whose implementation we will describe here, is capable of monitoring the state of the design as it evolves through the sequence of interactions between the human user and a front-end GUI. Before describing the implementation details of this back-end reasoning system, an overview of the implementation language will be presented in Section 4.1.

4.1 An Overview of Galileo

In this research the constraint programming language Galileo [7, 14] has been used as the modeling language of choice, since (a) is it one of the most expressive constraint programming languages, and (b) was developed specifically for exploitation in the engineering design domain. Galileo is a frame-based constraint programming language based on the First-Order Predicate Calculus (FOPC). Galileo offers designers a rich language for describing the structure of a product, the environment in which a product is being developed and the responsibilities of the various participants in an integrated product development environment [7, 14, 37]. The inference capabilities of the Galileo run-time environment are based on arc-consistency [32], a limited form of path-consistency and propagation of known states. The Galileo run-time environment is capable of offering justifications and explanations for domain reductions and conflicts [6].

A frame-based constraint programming language provides a designer with the expressiveness required to describe the various aspects of the design problem effectively. Frames can be used to represent the product being designed, the components from which it is configured or the materials from which it is made. Frames can also be used to describe the life-cycle environment in which the product will be manufactured, tested and deployed. Constraints between frames can be used to express the mutual restrictions between the objects in the design and the product's functionality, the component/material properties and the product life-cycle.

Amongst the many features of the Galileo language are the availability of predefined domains such as the real and integer numbers, arbitrary scalars and frame-like structured domains. It is possible to define sets and sequences in Galileo and structured domains can be defined and organized into inheritance hierarchies. The language comes with a number of predefined predicates such as equality and numeric inequality. There are a number of standard functions available which includes the complete range of arithmetic and trigonometric functions. There are also a number of set- and sequence-based predicates and functions available as standard. Compound constraints can be written using the standard logic connectives as well as the negation operator.

In Galileo, constraints can be universally or existentially quantified. Furthermore, quantifiers can be nested arbitrarily. In addition the existence of certain parameters and constraints can be expressed as being dependent on certain conditions being met. This is due to the fact that Galileo is based on a generalization of First-Order Logic known as First-Order Free Logic [8]. This is the means by which Dynamic Constraint Satisfaction Problems [33] can be easily modeled in the language.

Using Galileo in conjunction with an interactive constraint filtering system (run-time environment), a reasoning technology for supporting interactive conceptual design can be built. We will demonstrate the implementation details of this system later in this section. However, a brief discussion of the characteristics of constraint filtering will be presented first.

Constraint filtering is a form of constraint processing which progressively restricts the ranges of possible values for parameters, by enforcing the restrictive effects of constraints. This means that a constraint network can capture the impact of a decision on the various parameters in the network. Suppose, for example, that a network represents the mutually constraining influences that exist between the functionality and production cost of a product. Such a network could, with equal ease, determine the impact of functionality decisions on production cost or, on the other hand, determine the impact of cost decisions on functionality. In other words, constraint filtering relies on the multi-directional inference properties of constraints to propagate the consequences of decision throughout a constraint network.

In the following section important aspects of the constraint-based implementation of the design theory presented in Section 3 will be presented. When implementing this theory, a distinction can be made between concepts which are common across all design applications and concepts which are company-, application- or domain-specific. In Section 4.2 we will describe the generic aspects of the implementation. In Section 5 an example of the approach will be discussed in the context of an interactive design scenario. As part of that discussion the implementation of more application-specific concepts will be highlighted where relevant. For a full discussion of the work being reported here, the reader is encouraged to refer to some of the existing literature which describes these concepts in far greater detail [35].

4.2 Implementing Generic Concepts

In Figure 3 the Galileo model of a generic scheme is presented. This shows that the concept of a scheme is implemented as a Galileo structured domain called `scheme` which has two fields, called `scheme_name` and `structure`, respectively.

Since a scheme exists solely to provide the functionality required in the design specification, its structure should be the embodiment of that functionality. This is reflected by Line 3 in Figure 3, where the `structure` field of a `scheme` is declared to be of type `embodiment`. This model is based of the fact that the designer is mostly concerned with producing *embodiments* for *intended functions* by *choosing*, from among the *known means*, those which will provide the required functionality. This is reflected in Figure 4 which presents the Galileo implementation of an `embodiment` as a structured

```
domain scheme
    =::= ( scheme_name : string,
           structure   : embodiment ).
```

Figure 3: The representation of a generic scheme.

domain which has four fields: `scheme_name`, `intended_function`, `chosen_means` and `reasons`.

```
1 domain embodiment
2     =::= ( hidden scheme_name : string,
3            intended_function  : func,
4            chosen_means       : known_means,
5            reasons            : set of func_id ).
```

Figure 4: Modeling the embodiment of a function.

The `scheme_name` field, of type `string`, cross-references an embodiment to the scheme to which it belongs; this field is marked as `hidden` so that it does not appear on the user-interface used in Section 5. The field `intended_function` represents the function which is to be provided by the embodiment; in Line 3 it is declared to be of type `func`. We will now consider the definition of type `func` in some detail, because when we have done so it will be easier to explain the rest of the `embodiment` definition.

First, it should be noted that the same type of functionality is frequently needed in different parts of a scheme; that is, the function to be provided by one embodiment may be the same type of function as that to be provided by a different embodiment in the same scheme (or, indeed, by an embodiment in a different scheme). Thus, a `func` must represent, not a function, but an *instance* of a function. Furthermore, of course, one function instance must be distinguishable from a different instance of the same function. Having noted this, consider the definition of a `func` provided in Figure 5.

```
1 domain func
2     =::= ( verb : string,
3            noun : string,
4            id   : func_id ).
```

Figure 5: Modeling a function instance in Galileo.

The approach to representing functionality is a symbolic one, consisting of representing a function by a verb-noun pair. As can be seen in Figure 5, this approach is implemented in the first two fields of the structured domain used to represent a `func`. Since a `func` is a function *instance*, it must contain some field which distinguishes it from other instances of the same function. On Line 4 in Figure 5, it can be seen that the approach used was to give each `func` an `id` field, of type `func_id`, which is a synonym for a unique positive integer.

Now let us return to the remaining part of the definition of an `embodiment`, which was presented in Figure 4. The third field in this structured domain is `chosen_means`. This represents the approach chosen by the designer to provide the `intended_function` for the `embodiment`. The repertoire of technologies known to, and approved by, a company varies, of course, from one company to another. Thus, the definition of `known_means` is not generic; it depends on the design domain to which the conceptual design advice system is being applied.

Before proceeding to discuss the final field in an `embodiment`, consider the constraint shown in Figure 6. This specifies that whatever `known_means` is chosen for an embodiment must, in fact, be capable of providing the function intended for the embodiment.

The definition of the relation `can_be_used_to_provide` states that a `known_means` can provide a

```
all embodiment(E):
    can_be_used_to_provide(E.chosen_means, E.intended_function).

relation can_be_used_to_provide( known_means, func)
    =::= { (K,F): can_simultaneously_provide(K,Fs) and F in Fs}.
```

Figure 6: The means chosen for an embodiment must be valid.

function if that function appears in some set of functions that the means can simultaneously provide. The relation `can_simultaneously_provide` used in this definition is application-specific knowledge.

The final field in the definition of an `embodiment` (Figure 4) is called `reasons`. An embodiment may be introduced into a scheme because of different factors: it could be introduced to provide the top level functionality required in the design specification; alternatively, it could be introduced to provide some functionality whose necessity was recognized when some design principle was used during the development of the scheme. The `reasons` field in an `embodiment` records the motivation for introducing the `embodiment`. It does so by recording the identity numbers of the function instances whose provision required the introduction of the embodiment. This is reflected by the fact that, in Line 5 in Figure 4, the `reasons` field has the type `set of func_id`. The `reasons` field of an `embodiment` provides the basis for identifying those design entities between which context relations must be considered.

4.2.1 A Generic Model of Means

Figure 7 illustrates how the generic notion of a means can be modeled. As shown in Lines 1–4, a means is implemented as a Galileo structured domain called `means`. This has three fields: `scheme_name`, `type` and `funcs_provided`. As was the case with the definition of an embodiment, the `scheme_name` field is used to cross-reference a `means` with the scheme in which the `means` is being used.

```
1 domain means
2     =::= ( hidden scheme_name : string,
3            type               : means_type,
4            funcs_provided     : set of func_id ).

5 all means(M):
6     is_a_possible_behaviour_of( M.funcs_provided, M ).

7 domain means_type
8     =::= { a_principle, an_entity }.
```

Figure 7: Modeling a design means in Galileo

Note that there are two kinds of means: principles and entities. This is reflected by the fact that, as shown in Figure 7, the domain from which the `type` field of a `means` takes its value contains only two possible values, `a_principle` and `an_entity` (Lines 7–8).

The final field in the definition of the generic notion of a `means` is called `funcs_provided` and is of type `set of func_id`. It is used to remember which function instances within a scheme the `means` is being used to provide. Of course, a `means` should be used to provide only those function instances which it is capable of providing; this requirement is captured in the constraint in Lines 5–6. The definition of the relation `is_a_possible_behaviour_of` is an application-specific concept.

Based on the generic notion of a means, generic definitions for design principles and design entities can be defined. The generic notion of a `principle` and an `entity` are defined in Figure 8 as specializations of the generic notion of a `means`.

```
domain principle
    =::= { P: means(P) and
              P.type = a_principle }.

domain entity
    =::= { E: means(E) and
              E.type = an_entity }.
```

Figure 8: Generic design principle and design entity models.

4.2.2 Context Relationships and Entity Interfaces

As seen earlier, a design principle introduces a set of embodiments and a set of context relationships between them. Eventually, of course, each embodiment is realized by the introduction of design entity instances. This means that context relationships between embodiments will have to be realized by interfacing appropriately the entity instances which realize the embodiments. The details of these interfaces, which must be known in order to evaluate the quality of the scheme being developed, are represented as specializations of a generic concept called an `interface`, whose definition is provided in Figure 9.

```
1  domain interface
2      =::= ( hidden scheme_name : string,
3             entity_1 : entity_id,
4             entity_2 : entity_id ).

5  all interface(I):
6      exists entity(E1), entity(E2):
7          I.entity_1 = E1.id and
8          I.entity_2 = E2.id and
9          is_in_the_same_scheme_as( I, E1 ) and
10         is_in_the_same_scheme_as( I, E2 ).
```

Figure 9: Modeling generic interfaces between design entities.

As can be seen in Lines 1–4, an `interface` is defined between a pair of entities. The constraint defined in Lines 5–10 ensures that, for every `interface` which is defined, both of the entity instances to which it relates exist in the same scheme as the interface.

We shall see later how this generic definition of an interface can be used to define application-specific interfaces for embodying context-relations.

4.2.3 Generic Concepts for Comparing Schemes

A design specification may include several preferences. The basic notion in the approach for comparing schemes is the *preference*. It is defined in Figure 10 as a structured domain containing two fields: the `value` field, which contains the value of whatever scheme property is the subject of the preference, and the `intent` field, which indicates whether it is preferred that this scheme property be minimized or maximized.

When two schemes are being compared, this will involve comparing how well they perform with respect to each preference given in the design specification. A relation called `better_than` is used for comparing the instantiation of a preference in one scheme with the instantiation of the same preference in the other scheme. The definition of this relation is given in Figure 11. It can be seen that, if a preference involves minimizing some property, the better instantiation of the preference is the one with

```
domain intention
    =::= { minimal, maximal }.

domain preference
    =::= ( value  : real,
           intent : intention ).
```

Figure 10: Modeling a design preference.

the smaller value; similarly, if a preference involves maximizing some property, the better instantiation of the preference is the one with the larger value.

```
relation better_than( preference, preference )
    =::= { (P1,P2): P1.intent = minimal and
                    P2.intent = minimal and P1.value < P2.value,
           (P1,P2): P1.intent = maximal and
                    P2.intent = maximal and P1.value > P2.value }.
```

Figure 11: Comparing two design preference values to determine which is better.

As discussed in Section 3, no scheme for a product should dominate (in the sense of Pareto optimality) another scheme. This requirement is implemented as the constraint in Figure 12. If a designer develops a scheme which is dominated by a scheme that has been previously developed, this constraint will be violated and she will be given a message to that effect. Similarly, if a scheme is developed which dominates a previously developed scheme the constraint will be violated. In either case, it is intended that, as a result of the violation message, the designer will be motivated to improve the inferior scheme or else to discard it.

```
alldif scheme(S1), scheme(S2):
   not dominates( S1, S2 ).

relation dominates( scheme, scheme)
    =::= { (S1,S2): improves_on( S1, S2) and
                    not improves_on( S2, S1 ) }.
```

Figure 12: Comparing two schemes.

The constraint in Figure 12 is defined in terms of a relation called `dominates` which is also defined in that figure. We can see that one scheme dominates another scheme if the first scheme `improves_on` the latter (in respect of some preference) while at the same time it is not true that the latter scheme `improves_on` the first in respect of any preference.

The relation `improves_on`, between two schemes, is defined in terms of the relation `better_than`, between instantiations of preferences. However, this definition is not generic; it will vary from one design specification to another. Therefore, it is an application-specific concept.

5 An Example of Interactive Conceptual Design

In this section an example conceptual design problem is presented. The design problem considered is based on vehicle design. The presentation of this design problem comprises three phases. Firstly, in Section 5.1, the design specification will be discussed. Secondly, in Section 5.2, an appropriate constraint-based design knowledge-base will be presented. Finally, in Section 5.3, the development of

```
1  domain vehicle_scheme
2      =::= { S: scheme( S ) and

3                provides_the_function( S.structure.intended_function,
4                                     'provide', 'transport' ) and

5                recyclable(S) and

6                exists( S.width : real ) and
7                !S.width = width_of( S ) and
8                S.width =< 2.0 and

9                exists( S.mass  : preference ) and
10               S.mass.intent = minimal and
11               !S.mass.value  = mass_of( S ) and

12               exists( S.number_of_parts : preference ) and
13               S.number_of_parts.intent = minimal and
14               !S.number_of_parts.value  = number_of_parts_in( S ) }.
```

Figure 13: Incorporating design preferences in the design specification model.

two schemes based on the design specification will be presented. The use of constraint filtering in the process of developing these schemes will be described using a number of screen-shots from a constraint filtering system that would be capable of reasoning about the extended version of the Galileo language that was proposed during this research.

5.1 The Design Specification

Each requirement in the design specification can be regarded as a constraint on the schemes that the designer will develop. The requirements defined in the design specification can be modeled as constraints which are universally quantified over all instances of the scheme representation. For example, consider the following design specification:

> Design a product which exhibits the following properties:
>
> - provides the function *provide transport*;
> - is *recyclable*;
> - has a *width not greater than 2m*;
> - has *minimal mass* and
> - comprises a *minimal number of parts*.

In this specification there is one functional requirement and four physical requirements. The functional requirement states that the product must provide the function *provide transport*. The physical requirements state that the product must be *recyclable*, must have a *width not greater than 2m*, should have *minimal mass* and should comprise a *minimal number of parts*. A constraint-based model of this design specification is presented in Figure 13.

Briefly, this model extends the generic concept of a scheme to include constraints reflecting the requirements of the design specification. Lines 3–4 state that the product being designed should be able to provide the function `provide transport`. Line 5 states that the product should be `recyclable`. This requirement is stated using a relation whose implementation could ensure that the materials used in each of the design entities in the scheme are themselves recyclable. The categorical physical requirement that the `width` be less than 2m is defined on Lines 6–8. The value of the `width` of the

```
relation has_better_mass_than( vehicle_scheme, vehicle_scheme )
    =::= { (S1,S2): better_than( S1.mass, S2.mass ) }.

relation has_better_number_of_parts_than( vehicle_scheme, vehicle_scheme )
    =::= { (S1,S2): better_than( S1.number_of_parts, S2.number_of_parts ) }.

relation improves_on( vehicle_scheme, vehicle_scheme )
    =::= { (S1,S2): has_better_mass_than( S1, S2 ) or
                    has_better_number_of_parts_than( S1, S2 ) }.
```

Figure 14: Determining when one scheme improves on another.

scheme is computed using the function `width_of`, which is part of the application-specific knowledge of the organization. There are two design preferences, one related to mass (Lines 9–11), the other to the number of parts in the scheme (Lines 12–14). The `intent` of these preferences is that their `value` be `minimal`. Their values being computed with the functions `mass_of` and `number_of_parts_in`, respectively. Both of these function are part of the application-specific knowledge of the organization concerned.

In order to compare the schemes that are generated by the designer, the application-specific relation `improves_on`, referred to in Figure 12, must be defined. The definition of this relation is presented in Figure 14. It can be seen that the relation is defined in terms of the design preferences on the `mass` and `number_of_parts` in the scheme. The relations `has_better_mass_than` and `has_better_number_of_parts_than` is defined in terms of the `better_than` relation which was presented earlier.

5.2 An Example Design Knowledge-Base

In Figure 15 an illustration of the means contained in an example design knowledge-base is presented. This knowledge-base comprises one design principle, called *bicycle*, and a number of design entities, such as a *wheel assembly* and a *saddle*. The set of behaviours for each means in the knowledge-base are presented under the icon representing the means. Recall that behaviour is a *set* of functions that the means can provide simultaneously (Section 3.2.2). Thus the behaviours for a means is a set of sets. Most of the means in this example knowledge-base have only one behaviour; that is, the set of behaviours for each means contains only one set of functions. Furthermore, most of the behaviours of the means in this knowledge-base can provide only one function at a time. However, the *molded frame* and *axle* design entities have more complex behaviours. The *molded frame* entity can provide two functions simultaneously: *provide support* and *support passenger*. The *axle* entity has two behaviours: it can provide the two functions *support wheel* and *facilitate rotation* simultaneously, but can also be used to provide the single function *punch holes*.

5.3 Scheme Generation

Once a constraint-based model of the design specification has been developed, the designer (or team of designers) has the task of developing a set of alternative schemes for the required product. The constraint-based model of the design specification contains constraints relating to the following issues:

- constraints based on the functional requirements of the product as stated in the design specification;
- constraints based on the categorical physical requirements of the product, or its life-cycle, as stated in the design specification;
- constraints based on the preferences regarding the values of particular design properties.

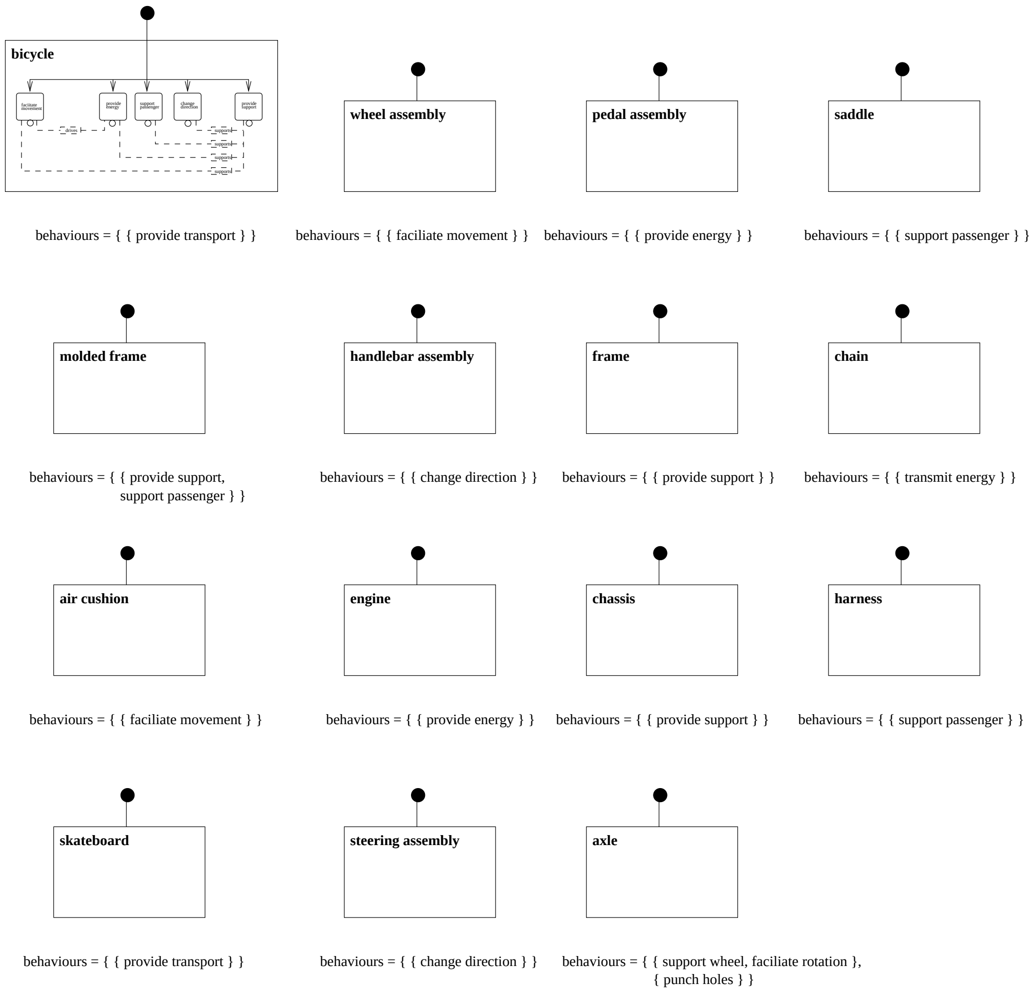

Figure 15: The means contained in an example design knowledge-base and their possible functionalities.

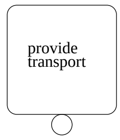

Figure 16: The functional requirement for a product.

Conceptual Design Adviser System – designer_1	
[ ] scheme_1.scheme_name	*'my vehicle'*
[ ] scheme_1.structure.intended_function.verb	provide
[ ] scheme_1.structure.intended_function.noun	transport
[ ] scheme_1.structure.intended_function.id	0
[ ] scheme_1.structure.chosen_means	_
[ ] scheme_1.structure.reasons	{}
[ ] scheme_1.width	_
[ ] scheme_1.mass.value	_
[ ] scheme_1.mass.intent	minimal
[ ] scheme_1.number_of_parts.value	_
[ ] scheme_1.number_of_parts.intent	minimal
> expand scheme_1.structure	
> expand scheme_1.structure.intended_function	
> expand scheme_1.mass	
> expand scheme_1.number_of_parts	

Figure 17: Examining the initial state of the scheme.

The designer's task is to develop a number of alternative schemes which satisfy the design specification. However, from a constraint-processing point of view, their task is to search for a set of schemes which satisfy the constraint-based design specification representation, each scheme resulting from making different choices among the various means for providing the required functionality. This process has already been discussed in Section 3.

The selection of means for providing the required functionality is subject to the various constraints in the design specification. For example, if a designer selects a means to embody a particular function which is not capable of providing the required functionality this violates the constraint shown in Figure 6.

As the designer selects a means for providing a particular function, further constraints are introduced into the scheme being developed, since each means has an associated set of constraints defining its properties. In this way the constraint-based model of the design comprises constraints representing the requirements stated in the design specification as well as constraints on functionality, scheme structure and life-cycle issues. The only restriction on the order in which the designer makes decisions is that design entities can only be introduced to embody functions that exist in the scheme. The degree to which the designer wishes to specify values for the parameters of design entities and interfaces before completing the function decomposition is essentially a matter for the designer.

In the following discussion of how the approach presented here supports interactive conceptual design a number of illustrative devices will be used. Firstly, the decisions that the designer makes will be illustrated through the use of diagrams. Secondly, at various critical points in the evolution of a scheme a text-based "screen-shot" will be presented to illustrate a number of critical aspects of the evolution of its constraint-based model.

The functional requirement for our scheme is illustrated in Figure 16. The functional requirement is to "*provide transport*". The initial constraint-based model of the scheme being developed here is illustrated in Figure 17.

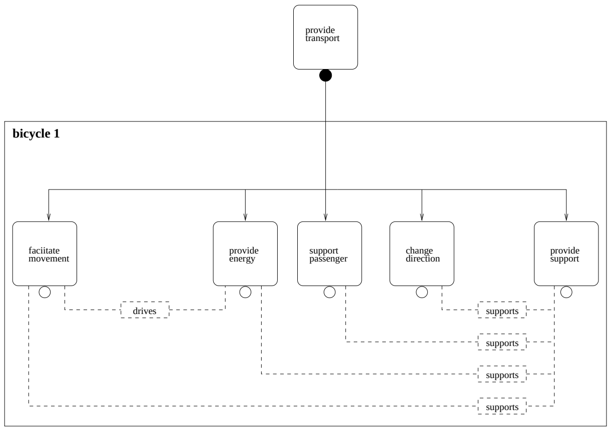

Figure 18: Using a design principle to embody the functional requirement.

It can be seen from Figure 17 that the `intended_function` of the `structure` of `scheme_1` is to `provide transport`. This function was specified in the definition of the `vehicle_scheme`. This function has an identifier (`id`) of 0 and the empty set as its set of `reasons`. This reflects the fact that this function was introduced into the scheme due to it being required by the design specification; in other words, no other function was responsible for this function being introduced into the scheme. The assignment of the 0 identifier and the empty set of reasons is done by a constraint, quantified over all schemes [35]. It can also be seen from this figure that the `mass` and `number_of_parts` associated with `scheme_1` are preferences. It can be seen that, in both cases, the `intent` is that these should have `minimal` values.

In Figure 18 an instance of the design principle *bicycle*, called *bicycle 1*, has been used to embody the function *provide transport*. This design principle introduces the need for five more functions to be embodied. The designer must now select means for embodying each of these functions. The presence of these additional embodiments is due to the constraint-based description of the bicycle design principle.

Figure 19 depicts the state of the constraint model of the scheme after the designer has selected `a_bicycle` as the `chosen_means` for providing the `intended_function` of the `structure` of `scheme_1`. The effect of this is that a new parameter called `bicycle_1` is automatically introduced[1]. The parameter `bicycle_1` is an instance of an application-specific design principle. Application-specific design principles can be defined as specializations of the generic design principle (Figure 8). The principle of a bicycle is defined in Figure 20.

This application-specific principle is defined to be a specialization of the generic notion of a principle (Line 2); the specialization is specified by the extra properties that are defined in Lines 3–22.

It was seen in Figure 2 that a `bicycle` principle involves five `embodiments`. These are specified in Lines 3–7 of Figure 20. The functions which Figure 2 states are to be provided by these embodiments (`facilitate movement`, `provide energy`, `support passenger`, `change direction` and `provide support`) are specified in Lines 8–17 of Figure 20.

The context relationships between the embodiments which are shown in Figure 2 are stated in Lines 18–22 of Figure 20. For example, in Line 18 it is stated that a `drives` relationship must exist between the `embodiment e2` and `embodiment e1`. Ultimately, as was discussed in Section 3, each embodiment introduced by a principle is realized by the introduction of a set of one or more design entity instances. We will see how these context relations affect interactive conceptual design later in this discussion.

While the designer could continue to develop the function decomposition of the scheme by employing more design principles, we will assume that the designer will proceed by embodying each function with a design entity. In Figure 21 the designer selects the *wheel assembly* design entity to embody

[1] Structured fields are indicated on the screen by the presence of a ▷, which is intended to "invite" the user to examine the field further by expanding it; the value of a scalar field whose value is known is shown; for a scalar field whose value is not yet known a "_" is shown.

Conceptual Design Adviser System – designer_1	
[ ] scheme_1.scheme_name	*'my vehicle'*
[ ] scheme_1.structure.intended_function.verb	provide
[ ] scheme_1.structure.intended_function.noun	transport
[ ] scheme_1.structure.intended_function.id	0
[ ] scheme_1.structure.chosen_means	a_bicycle
[ ] scheme_1.structure.reasons	{}
[ ] scheme_1.width	_
[ ] scheme_1.mass	▷
[ ] scheme_1.number_of_parts	▷
[ ] bicycle_1	▷

> scheme_1.structure.chosen_means = a_bicycle
Info: Parameter bicycle_1 has been created...

Figure 19: Using the principle of a bicycle in scheme_1 to provide the function “provide transport”.

```
domain bicycle
    =::= { B: principle(B) and
           exists( B.e1 : embodiment ) and
           exists( B.e2 : embodiment ) and
           exists( B.e3 : embodiment ) and
           exists( B.e4 : embodiment ) and
           exists( B.e5 : embodiment ) and
           provides_the_function( B.e1.intended_function,
                                  'facilitate', 'movement' ) and
           provides_the_function( B.e2.intended_function,
                                  'provide', 'energy' ) and
           provides_the_function( B.e3.intended_function,
                                  'support', 'passenger' ) and
           provides_the_function( B.e4.intended_function,
                                  'change', 'direction' ) and
           provides_the_function( B.e5.intended_function,
                                  'provide', 'support' ) and
           drives( B.e2, B.e1 ) and
           supports( B.e5, B.e1) and
           supports( B.e5, B.e2) and
           supports( B.e5, B.e3) and
           supports( B.e5, B.e4) }.
```

Figure 20: Definition of a company-specific design principle.

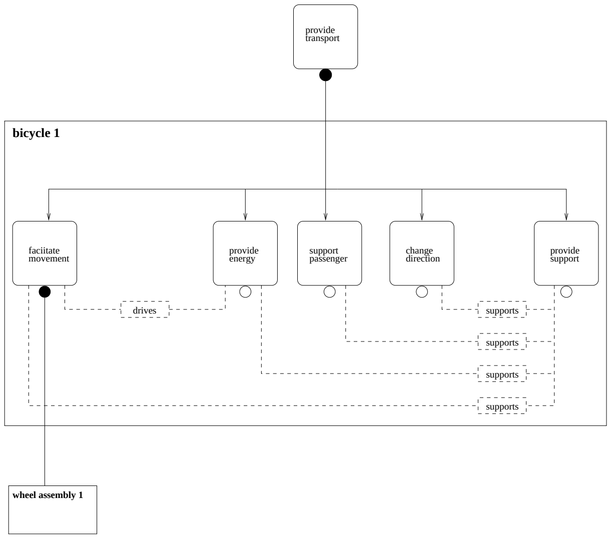

Figure 21: Using a design entity to embody a function in the scheme.

the function *facilitate movement*. This introduces an instance of this means, called *wheel assembly 1*, into the scheme. As the designer introduces design entities into the scheme the context relations that exist between the function embodiments must be considered. However, since there is only one design entity in the scheme presented in Figure 21 no context relations are considered at this point in the scheme's development.

In Figure 22 the state of the constraint model of the designers scheme is presented. In this figure the designer begins to explore the parameter `bicycle_1`. It can be seen that `bicycle_1` is a design principle and that the `funcs_provided` by this principle is a singleton set containing the value 0; this means that `bicycle_1` provides one function, namely, the function whose `id` is 0 – this was seen in Figure 17 to be the function required in the design specification. It can also be seen from Figure 22 that `bicycle_1` contains a number of other structured fields, namely, `e1`, `e2`, `e3`, `e4` and `e5`. These fields represent further embodiments which the designer must make in order to properly incorporate the `bicycle_1` design principle into `scheme_1`.

Conceptual Design Adviser System – `designer_1`	
[ ] `bicycle_1.type`	`a_principle`
[ ] `bicycle_1.funcs_provided`	{0}
[ ] `bicycle_1.e1.intended_function.verb`	`facilitate`
[ ] `bicycle_1.e1.intended_function.noun`	`movement`
[ ] `bicycle_1.e1.intended_function.id`	1
[ ] `bicycle_1.e1.chosen_means`	`a_wheel_assembly`
[ ] `bicycle_1.e1.reasons`	{0}
[ ] `bicycle_1.e2`	▷
[ ] `bicycle_1.e3`	▷
[ ] `bicycle_1.e4`	▷
[ ] `bicycle_1.e5`	▷
[ ] `wheel_assembly_1`	▷
> bicycle_1.e1.chosen_means = a_wheel_assembly	
Info: Parameter `wheel_assembly_1` has been created...	

Figure 22: Incorporating a `wheel_assembly` entity to provide the function "`facilitate movement`".

In Figure 22 the designer explores the parameter `bicycle_1.e1` by expanding it. It can be seen that the function to be embodied is `facilitate movement`. This function has an `id` of 1 since this is the next unique function identifier for this scheme. The `reasons` for the embodiment `bicycle_1.e1` is the singleton set containing the function identifier 0; this represents the fact that the function whose identifier is 0 is a reason for this embodiment. The designer chooses `a_wheel_assembly` as the `chosen_means` for this embodiment. This causes the automatic introduction of another new parameter, `wheel_assembly_1`, into the scheme.

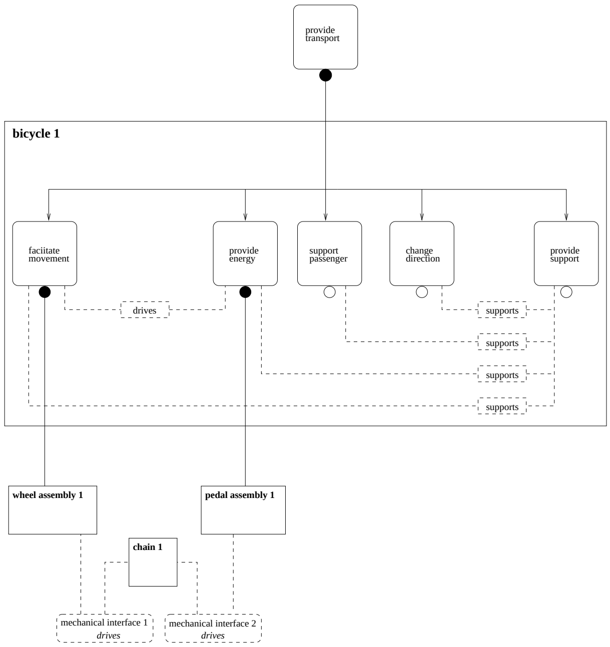

Figure 23: The effect of a context relation on the configuration of design entities.

In Figure 23 the designer has chosen to embody the function *provide energy* with the *pedal assembly* design entity. This introduces an instance of this means, called *pedal assembly 1*, into the scheme. Since the *drives* context relation must exist between the embodiments of the functions *facilitate movement* and *provide energy*, this caused, in addition to the existence of the design entities *wheel assembly 1* and *pedal assembly 1*, the introduction of an instance of the *chain* design entity, called *chain 1*. Both of these interfaces are used, along with *chain 1*, to embody the *drives* relation that should exist between *wheel assembly 1* and *pedal assembly 1*.

In Figure 24 the effects, on the constraint model of the scheme, of the designer's decision to use `a_pedal_assembly` as the `chosen_means` to embody the function `provide energy` are illustrated. The first effect is that a new parameter, `pedal_assembly_1`, is introduced into the constraint model. Although it is not apparent in Figure 24, the parameter `pedal_assembly_1` is a design entity. If the designer were to expand `pedal_assembly_1` we would see that its `id` field contains the value 2, reflecting the fact that it is the second design entity to be incorporated into the scheme.

Also shown in this figure are the introductions of the new parameters `chain_1`, `mechanical_interface_1` and `mechanical_interface_2`. As already stated, these parameters exist in order to fulfill the context relation `drives` that must exist between the embodiments for the functions `provide power` and `facilitate movement`, as specified in the principle of a bicycle. The need for this context relation is due to the `bicycle` design principle used earlier in the interaction.

The meaning of the `drives` context relation is an application-specific concept. One possible definition for it is defined in Figure 25. It can be seen that a `drives` relationship holds between a pair of embodiments if there exists a `drives` relationship between the design entities which are used to provide the functionality associated with them.

The precise realization of the context relationship specified in a principle depends on which design entities are used to realize the embodiments that must satisfy the context relationship. Suppose that

Conceptual Design Adviser System – designer_1	
[ ] bicycle_1.type	a_principle
[ ] bicycle_1.funcs_provided	{0}
[ ] bicycle_1.e1	▷
[ ] bicycle_1.e2.intended_function.verb	provide
[ ] bicycle_1.e2.intended_function.noun	energy
[ ] bicycle_1.e2.intended_function.id	2
[ ] bicycle_1.e2.chosen_means	a_pedal_assembly
[ ] bicycle_1.e2.reasons	{0}
[ ] bicycle_1.e3	▷
[ ] bicycle_1.e4	▷
[ ] bicycle_1.e5	▷
[ ] wheel_assembly_1	▷
[ ] pedal_assembly_1	▷
[ ] chain_1	▷
[ ] mechanical_interface_1	▷
[ ] mechanical_interface_2	▷

```
> bicycle_1.e2.chosen_means = a_pedal_assembly
Info: Parameter pedal_assembly_1 has been created...
Info: Parameter chain_1 has been created...
Info: Parameter mechanical_interface_1 has been created...
Info: Parameter mechanical_interface_2 has been created...
```

Figure 24: Using a pedal assembly to provide the function "provide energy".

```
relation drives( embodiment, embodiment )
    =::= { (E1,E2): drives( { X | exists entity( X ):
                                          derives_from( X, E1 ) },
                            { Y | exists entity( Y ):
                                          derives_from( Y, E2 ) } ) }.

relation drives( set of entity, set of entity )
    =::= { (E1s,E2s): exists E1 in E1s, E2 in E2s: drives( E1, E2 ) }.

relation drives( entity, entity )
    =::= { (P,W): pedal_assembly(P) and wheel_assembly(W) and
                  is_in_the_same_scheme_as( P, W ) and
                  !exists chain(C):
                      is_in_the_same_scheme_as( P, C ) and
                      !exists mechanical_interface(M1):
                          M1.entity1 = P.id and
                          M1.entity2 = C.id and
                          M1.relationship = drives and
                      !exists mechanical_interface(M2):
                          M2.entity1 = W.id and
                          M2.entity2 = C.id and
                          M2.relationship = drives }.
```

Figure 25: The meaning of the drives context relation.

a `pedal_assembly` is the design entity used to `provide energy` and a `wheel_assembly` is the design entity used to `facilitate movement`, we can see in Figure 25 the relationship that would have to be satisfied between these two entity instances in order to properly embody the drives context relation. According to the definition of this relationship, if a `pedal_assembly` is to `drive` a `wheel_assembly`, they must be `in_the_same_scheme` and there must be a further design entity instance, a `chain`, in the same scheme. These entity instances must be interfaced in the following way: there must be a `mechanical_interface` between the `pedal_assembly` and the `chain` and another one between the `wheel_assembly` and the `chain`.

As we shall now see, a `mechanical_interface` is simply a specialization of the generic notion of an `interface` which we encountered in Section 4.2.2. The definition of a `mechanical_interface` is given in Figure 26. It can be seen to be a specialization of a application-specific notion of interface, called a `raleigh_interface`, which, from its definition in Figure 27, can be seen to be a specialization of the generic notion of `interface`.

```
domain mechanical_interface
    =::= { S: raleigh_interface(S) and S.type = mechanical and
              exists( S.relationship : mechanical_relationship ) }.

domain mechanical_relationship
    =::= { controls, drives, supports }.
```

Figure 26: Modeling a `mechanical` interface.

It can be seen from Figure 26 that a `mechanical_interface` is a `raleigh_interface` whose `type` field contains the value `mechanical` and which also has an additional field called `relationship` that specifies the nature of the mechanical relationship involved in the interface; it can be seen that three kinds of relationship are supported: `controls`, `drives` and `supports`.

It can be seen from Figure 27 that a `raleigh_interface` is simply an `interface` with an additional field called `type` which specifies the class of relationship involved in the interface; it can be seen that two classes of relationship are supported: `spatial` and `mechanical`.

```
domain raleigh_interface
    =::= { I: interface(I) and
              exists( I.type : raleigh_interface_type ) }.

domain raleigh_interface_type
    =::= { spatial, mechanical }.
```

Figure 27: Modeling company-specific interfaces.

Therefore, the parameter `chain_1` exists in order to satisfy the context relation that the embodiment for the function `provide power` drives the embodiment for the function `facilitate movement`. According to the application-specific definition of the `drives` relation there must be a `mechanical_interface` between `pedal_assembly_1` and `chain_1` and another between `wheel_assembly_1` and `chain_1`. This will be explored in further detail in Figure 28. (The parameter `chain_1` is the third design entity to be incorporated into this scheme; thus, if we were to expand it, we would see that its `id` field contains the value 3).

Figure 29 shows the state of the scheme after the designer has chosen to embody the function *support passenger* with the design entity *saddle*, the function *change direction* with the design entity *handlebar assembly* and the function *provide support* with the design entity *frame*. Due to the *bicycle* design principle, a context relation called *supports* must exist between the embodiment of the function *provide support* and the embodiments of each of the functions *facilitate movement*, *provide energy*, *support passenger* and *change direction*.

Conceptual Design Adviser System – `designer_1`	
[ ] `bicycle_1`	▷
[ ] `wheel_assembly_1`	▷
[ ] `pedal_assembly_1`	▷
[ ] `chain_1`	▷
[ ] `mechanical_interface_1.entity_1`	1
[ ] `mechanical_interface_1.entity_2`	3
[ ] `mechanical_interface_1.type`	`mechanical`
[ ] `mechanical_interface_1.relationship`	`drives`
[ ] `mechanical_interface_2`	▷

> contract bicycle_1
> expand mechanical_interface_1

Figure 28: Embodying the `drives` context relation.

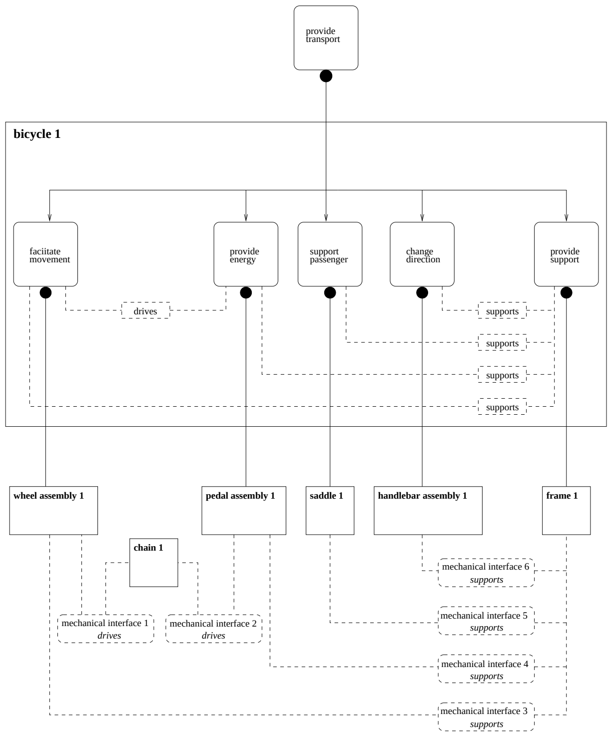

Figure 29: An example scheme configuration.

Conceptual Design Adviser System – designer_1	
[ ] scheme_1.scheme_name	*'my vehicle'*
[ ] scheme_1.structure	▷
[ ] scheme_1.width	_
[ ] scheme_1.mass.value	12
[ ] scheme_1.mass.intent	minimal
[ ] scheme_1.number_of_parts.value	6
[ ] scheme_1.number_of_parts.intent	minimal
[ ] bicycle_1	▷
[ ] wheel_assembly_1	▷
[ ] pedal_assembly_1	▷
[ ] chain_1	▷
[ ] mechanical_interface_1	▷
[ ] mechanical_interface_2	▷
[ ] frame_1	▷
[ ] mechanical_interface_3	▷
[ ] mechanical_interface_4	▷
[ ] saddle_1	▷
[ ] mechanical_interface_5	▷
[ ] handlebar_assembly_1	▷
[ ] mechanical_interface_6	▷
> expand scheme_1 > expand scheme_1.mass > expand scheme_1.number_of_parts	

Figure 30: The state of `scheme_1` once materials have been selected for all the entities from which it is configured.

Each of these context relations is embodied by a *mechanical interface* that defines a *supports* relationship. The details of these mechanical interfaces that define a *supports* relationship will be specified during detailed design. Since all the functions have been embodied in the scheme presented in Figure 29, the designer can focus on selecting values for the attributes associated with each design entity in the scheme. In making these decisions the designer must ensure that the various constraints that are imposed on her due to the design specification or the design knowledge-base must be satisfied. In addition, this scheme must be compared with any other alternative scheme for this product that are developed.

In Figure 30 the state of `scheme_1` is shown after several more decisions have been made by the designer, namely after materials have been selected for all the entities from which it is configured. In this figure, the `mass` and `number_of_parts` fields of `scheme_1` have been expanded. It can be seen that the total mass of this scheme is estimated to be 12 units and that it comprises 6 parts. These values are computed using a number of application-specific functions defined over sets of entities used in a particular scheme. Their definition is trivial to define, so it will not be considered further here.

In Figure 31 a second scheme is presented which, for simplicity, is based on the same embodiments used to develop `scheme_1`. In this figure the state of `scheme_2` after materials have been selected for `wheel_assembly_2` and `pedal_assembly_2` is illustrated. It can be seen that the mass of `scheme_2` is currently estimated to be 20 units and that it comprises 6 parts. Therefore, this scheme is certainly dominated by `scheme_1` since, although both schemes have the same number of parts as each other, `scheme_1` has the smaller mass. This means that, since `scheme_2` does not improve on `scheme_1` on any design preference, `scheme_2` is dominated by `scheme_1`. The designer's attention is drawn to this fact by the message stating that a constraint violation has been detected. In particular, in this

Conceptual Design Adviser System – `designer_2`	
[ ] `scheme_2.scheme_name`	*'another vehicle'*
[ ] `scheme_2.structure`	▷
[ ] `scheme_2.width`	_
[ ] `scheme_2.mass.value`	`20`
[ ] `scheme_2.mass.intent`	`minimal`
[ ] `scheme_2.number_of_parts.value`	`6`
[ ] `scheme_2.number_of_parts.intent`	`minimal`
[ ] `bicycle_2`	▷
[ ] `wheel_assembly_2`	▷
[ ] `pedal_assembly_2`	▷
[ ] `chain_2`	▷
[ ] `mechanical_interface_7`	▷
[ ] `mechanical_interface_8`	▷
[ ] `frame_2`	▷
[ ] `mechanical_interface_9`	▷
[ ] `mechanical_interface_10`	▷
[ ] `saddle_2`	▷
[ ] `mechanical_interface_11`	▷
[ ] `handlebar_assembly_2`	▷
[ ] `mechanical_interface_12`	▷

> wheel_assembly_2.material = steel
> pedal_assembly_2.material = steel
ATTENTION: the following constraint was violated:

```
alldif scheme(S1), scheme(S2):  not dominates(S1,S2)
```

> expand scheme_2
> expand scheme_2.mass
> expand scheme_2.number_of_parts

Figure 31: The state of `scheme_2` after materials have been selected for `wheel_assembly_2` and `pedal_assembly_2`.

case, the constraint which compares schemes to ensure that none are dominated is violated. This illustrates how, during an interactive design session, inconsistencies arising from designer decisions can be instantly reported for corrective action.

5.4 Review of the Example

In Section 5 a detailed discussion of how the approach described in this paper can be used to provide the underlying reasoning capability of an interactive conceptual design system was presented. The interaction between the front-end GUI, presented using diagrams, and the back-end constraint-based model, presented using text-based screen-shots, illustrates how the work presented here can be used as a basis for building interactive CAD systems to support conceptual design.

The research presented here has been validated in a number of industrial settings on a number of different design domains. For example, the work presented here has been used to develop conceptual designs for products in mechatronics, optical systems and electronic component design.

6 Comparison with Related Research

The approach to supporting conceptual design presented here is based on a combination of design theory, constraint processing techniques and Pareto optimality. In this section, this approach will be compared with a number of state-of-the-art approaches which have been reported in the literature. The approaches are categorized as being either design theory-driven (Section 6.1), constraint processing-driven (Section 6.2) or Pareto optimality-driven (Section 6.3).

6.1 Design Theory Approaches

The design theory upon which the approach presented here is based assumes that products exist to provide some required functionality. There are a number of theories of design, such as "The Theory of Domains" [1] and "The General Procedural Model of Engineering Design" [29], that describe the parallelism between the decomposition of a functional requirement and the composition of a set of parts that fulfill that requirement.

The *function-means tree* approach to design synthesis is one approach that assists the designer is decomposing a functional requirement into an equivalent set of functions that can be provided by a set of known parts [11]. A function-means tree describes alternative ways of providing a top-level (root) function through the use of means. A means is a known approach to providing functionality. Two types of means can be identified in a function-means tree: principles and entities. A principle is defined by a collection of functions which, collectively, provide a particular functionality; it carries no other information than the lower-level functions to be used in order to provide a higher-level function. An entity represents a part or sub-assembly.

In the approach adopted here, the function-means tree concept was extended by adding context relations between the functions that define a design principle. This enables a computer to assist a designer to reason about the configuration of a set of design entities that obey the relationships that should exist between the functions in a design. It also helps to ensure that there is a valid mapping between the functional decomposition of a product and its physical composition in terms of parts.

The Scheme-Builder system [9, 42] uses function-means trees as a basis for structuring a design knowledge-base and generating schemes. The system interprets a function as an input-output transformation. The advantage of the system is that it is very systematic in terms of how functions are decomposed into sets of equivalent functions. However, its applications are limited to very highly parameterized design domains, such as mechatronics and control systems. The symbolic approach to representing function adopted in the research presented here, coupled with the use of context relations in design principles, makes our approach far more flexible.

6.2 Constraint-Based Approaches

A number of systems have been developed for supporting aspects of conceptual design based on constraints. The "Concept Modeler" system was one of the earliest of such systems reported in the literature [46]. Aspects of the approach adopted in Concept Modeler were extended in a system called "Design Sheet" [10]. These systems focused on using constraint processing techniques to manage consistency within a constraint-based model of a design. In these systems conceptual designs are represented as systems of algebraic equations.

The approach presented in this paper addresses a wider variety of issues that are crucial to successful conceptual design. The most important of these issues is design synthesis. In the approach presented here a designer is assisted in interactively synthesizing a scheme for a design specification. In addition, a designer can develop multiple schemes for a design specification and be offered advice based on a comparison of these schemes. These are critical issues to supporting conceptual design which are not addressed in either Concept Modeler or Design Sheet.

The work presented in here builds on earlier work on interactive constraint processing for engineering design [7]. The earlier work focused on using constraint processing as a basis for interacting with a human designer who was working on a detailed model of design. The work presented here builds on this work by demonstrating that using constraint processing as the basis for interacting with a human designer can be extended to support the development of a number of alternative schemes for a design specification from an initial statement of functional and physical requirements.

6.3 Pareto Optimality Approaches

The principle of Pareto optimality has been applied to a wide variety of problems in design. Most of these applications have used the principle of Pareto optimality in conjunction with evolutionary algorithms to generate a set of "good" design concepts [12, 23, 39]. These approaches focus on the automatic generation of design alternatives, an issue not of interest in the research presented here.

The use of the principle of Pareto optimality to monitor progress in design has been reported [41]. The approach focuses on the "tracking" of Pareto optimality to co-ordinate distributed engineering agents. Tracking Pareto optimality, in this case, means that the problem solver being used can automatically recognize Pareto optimality loss and the particular opportunity to improve the design. That approach inspired aspects of the approach to using Pareto optimality in the research presented here. However, in this research, Pareto optimality is used to compare two different schemes for a design specification rather than recognizing when Pareto optimality is lost within an individual scheme. In this research, it is believed that the natural competition between designers can be harnessed to motivate improvements in the quality of schemes.

7 Conclusion

This paper presents an interactive constraint-based approach to supporting a human designer during engineering conceptual design. The approach is based upon an expressive and general technique for modeling: the design knowledge which a designer can exploit during a design project; the life-cycle environment which the final product faces; the design specification which defines the set of requirements that the product must satisfy; and the structure of the various schemes that are developed by the designer. A computational reasoning environment based on constraint filtering is proposed as the basis of an interactive conceptual design support tool. Using such a tool, the designer can be assisted in developing and evaluating a set of schemes which satisfy the various constraints that are imposed on the design.

The primary contribution of this research is that it provides a novel approach to supporting the interaction between the human designer and a constraint-based environment for conceptual design. The approach presented here not only addresses the issue of modeling and reasoning about the design of products from an abstract set of requirements, but it also demonstrates how life-cycle knowledge can be incorporated into the conceptual design of a product and how alternative schemes can be compared.

Acknowledgments

The author would like to acknowledge the feedback and advice contributed by Professor James Bowen (University College Cork, Ireland) during the research reported here. The author is funded by Enterprise Ireland through their Research Innovation Fund (Project Number RIF/2001/317).

References

[1] Mogens Myrup Andreasen. The Theory of Domains. In *Proceedings of Workshop on Understanding Function and Function-to-Form Evolution*, Cambridge University, 1992.

[2] Mogens Myrup Andreasen and Lars Hein. *Integrated Product Development*. IFS Publications Ltd./Springer-Verlag, Bedford, UK, 1987.

[3] D. Bahler, C. Dupont, and J. Bowen. An axiomatic approach that supports negotiated resolution of design conflicts in Concurrent Engineering. In *Artificial Intelligence in Design*, pages 363–379, 1994. Kluwer Academic Press, Netherlands.

[4] Sanjay Bhansali, Glenn A. Kramer, and Tim J. Hoar. A principled approach towards symbolic geometric constraint satisfaction. *Journal of Artificial Intelligence Research*, 4:419–443, June 1996.

[5] William P. Birmingham and Allen Ward. What is Concurrent Engineering? *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 9:67–68, 1995. Guest Editorial in a Special Issue on Concurrent Engineering.

[6] James Bowen. Using dependency records to generate design coordination advice in a constraint-based approach to Concurrent Engineering. *Computers in Industry*, 33(2–3):191–199, 1997.

[7] James Bowen and Dennis Bahler. Frames, quantification, perspectives and negotiation in constraint networks in life-cycle engineering. *International Journal for Artificial Intelligence in Engineering*, 7:199–226, 1992.

[8] James A. Bowen and Dennis Bahler. Conditional existance of variables in generalised constraint networks. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI)*, pages 215–220, 1991.

[9] R. H. Bracewell and J. E. E. Sharpe. Functional descriptions used in computer support for qualitative scheme generation = "Schemebuilder". *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):333–345, 1996.

[10] M. J. Buckley, K. W. Fertig, and D. E. Smith. Design sheet: An environment for facilitating flexible trade studies during conceptual design. In *AIAA 92-1191 Aerospace Design Conference*, February 1992. Irvine, California.

[11] J. Buur. *A Theoretical Approach to Mechatronics Design.* PhD thesis, Technical University of Denmark, Lyngby, 1990.

[12] Matthew I. Campbell, Jonathan Cagan, and Kenneth Kotovsky. A-design: Theory and implementation of an adaptive agent-based method of conceptual design. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '98*, pages 579–598, The Netherlands, July 1998. Kluwer Academic Publishers.

[13] Amaresh Chakrabarti and Lucienne Blessing. Guest editorial: Representing functionality in design. *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):251–253, 1996.

[14] Marc van Dongen, Barry O'Sullivan, James Bowen, Alex Ferguson, and Mike Baggaley. Using constraint programming to simplify the task of specifying DFX guidelines. In Kulwant S. Pawar, editor, *Proceedings of the 4th International Conference on Concurrent Enterprising*, pages 129–138, University of Nottingham, October 1997.

[15] A. H. B. Duffy, M. M. Andreasen, K. J. MacCallum, and L. N. Reijers. Design Co-ordination for Concurrent Engineering. *Journal of Engineering Design*, 4(4):251–265, December 1993.

[16] Boi Faltings and Eugene C. Freuder, editors. *IEEE Intelligent Systems and their applications*, volume 13. IEEE Computer Society, July–August 1998. Special Issue on Configuration.

[17] Yousri El Fattah. Constraint logic programming for structure-based reasoning about dynamic physical systems. *Artificial Intelligence in Engineering*, 1:253–264, 1996.

[18] Gerhard Fleischanderl, Gerhard Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems and their applications*, 13(4):59–68, July–August 1998. Special Issue on Configuration.

[19] M. J. French. *Engineering Design: The Conceptual Stage.* Heinemann Educational Books, London, 1971.

[20] Xiao-Shan Gao and Shang-Chou Chou. Solving geometric constraint systems I: A global propagation approach. *Computer-Aided Design*, 30(1):47–54, January 1998.

[21] Xiao-Shan Gao and Shang-Chou Chou. Solving geometric constraint systems II: A symbolic approach and decision of Rc-constructibility. *Computer-Aided Design*, 30(2):115–122, February 1998.

[22] Esther Gelle and Ian Smith. Dynamic constraint satisfaction with conflict management in design. In Michael Jampel, Eugene Freduer, and Michael Maher, editors, *OCS'95: Workshop on Over-Constrained Systems at CP'95*, pages 33–40, Cassis, Marseilles, Sep 1995.

[23] J. S. Gero and S. Louis. Improving pareto optimal designs using genetic algorithms. *Microcomputers in Civil Engineering*, 10(4):241–249, 1995.

[24] S. R. Gorti, S. Humair, R. D. Sriram, S. Talukdar, and S. Murthy. Solving constraint satisfaction problems using ATeams. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 10:1–19, 1995.

[25] Mark D. Gross, Stephen M. Ervin, James A. Anderson, and Aaron Fleisher. Constraints: Knowledge representation in design. *Design Studies*, 9(3):133–143, July 1988.

[26] Djamila Haroud, Sylvie Boulanger, Esther Gelle, and Ian Smith. Management of conflict for preliminary engineering design tasks. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 9:313–323, 1995.

[27] W. Hsu and B. Liu. Conceptual design: Issues and challenges. *Computer-Aided Design*, 32(14):849–850, 2000.

[28] Wynne Hsu and Irene M. Y. Woon. Current research in the conceptual design of mechanical products. *Computer-Aided Design*, 30(5):377–389, 1998.

[29] Vladimir Hubka and W. Ernst Eder. *Engineering Design: General Procedural Model of Engineering Design*. Serie WDK - Workshop Design-Konstruktion. Heurista, Zurich, 1992.

[30] C. Lottaz, I.F.C. Smith, Y. Robert-Nicoud, and B.V. Faltings. Constraint-based support for negotiation in collaborative design. *Artificial Intelligence in Engineering*, 14:261–280, 2000.

[31] C. Lottaz, R. Stalker, and I. Smith. Constraint solving and preference activation for interactive design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12:13–27, 1998.

[32] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[33] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI 90, Eighth National Conference on Artificial Intelligence*, volume 1, pages 25–32, Boston, MA, July–August 1990. AAAI Press, Menlo Park, CA, U.S.A.

[34] Y. Nagai and S. Terasaki. A constraint-based knowledge compiler for parametric design problem in mechanical engineering. Technical Report TM-1270, ICOT, Japan, June 1993.

[35] Barry O'Sullivan. *Constraint-Aided Conceptual Design*. PhD thesis, Department of Computer Science, University College Cork, Ireland, July 1999. (Also published by Professional Engineering Publishing, 2001, ISBN: 1-86058-335-0).

[36] Barry O'Sullivan. Constraint-based product structuring for configuration. In *ECAI-2002 Workshop on Configuration*, July 2002.

[37] Barry O'Sullivan, James Bowen, and Alexander B. Ferguson. A new technology for enabling computer-aided EMC analysis. In *Workshop on CAD Tools for EMC (EMC-York 99)*, July 1999.

[38] G. Pahl and W. Beitz. *Engineering Design: A systematic approach.* Springer, London, 2nd edition, 1995.

[39] Ian C. Parmee. Adaptive search techniques for decision support during preliminary engineering design. In *Proceedings of Informing Technologies to Support Engineering Decision Making*, EPSRC/DRAL Seminar, Institution of Civil Engineers, London, November 1994.

[40] Charles Petrie, Heecheol Jeon, and Mark R. Cutkosky. Combining constraint propagation and backtracking for distributed engineering. In Walter Hower and Ruttkay. Zsòfia, editors, *Workshop on Non-Standard Constraint Processing, ECAI 96*, pages 84–94, August 1996. Budapest, Hungary.

[41] Charles J. Petrie, Teresa A. Webster, and Mark R. Cutkosky. Using Pareto optimality to coordinate distributed agents. *AIEDAM*, 9:269–281, 1995.

[42] I. Porter, J. M. Counsell, and J. Shao. Knowledge representation for mechatronic systems. In Alan Bradshaw and John Counsell, editors, *Computer-Aided Conceptual Design '98*, pages 181–195. Lancaster University, May 1998. Proceedings of the 1998 Lancaster International Workshop on Engineering Design.

[43] S. Y. Reddy, K. W. Fertig, and D. E. Smith. Constraint management methodology for conceptual design tradeoff studies. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences and Computers in Engineering Conference*, August 1996. Irvine, California.

[44] Daniel Sabin and Eugene C. Freuder. Configuration as composite constraint satisfaction. In *AAAI-96 Fall Symposium on Configuration*, pages 28–36, 1996. also in: Proceedings, Artificial Intelligence and Manufacturing Research Planning Workshop 1996.

[45] Daniel Sabin and Rainer Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems and their applications*, 13(4):42–49, July–August 1998. Special Issue on Configuration – Getting it Right.

[46] David Serrano. *Constraint Management in Conceptual Design.* PhD thesis, Massachusetts Institute of Technology, October 1987.

[47] Shuichi Shimizu and Masayuki Numao. Constraint-based design for 3D shapes. *Artificial Intelligence*, 91:51–69, 1997.

[48] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing*, 12(4):357–372, 1998.

[49] Anna C. Thorton. A support tool for constraint processes in embodiment design. In T. K. Hight and F. Mistree, editors, *ASME Design Theory and Methodology Conference*, pages 231–239, Minneapolis, USA, September 1994.

[50] Marcel Tichem. *A Design Coordination Approach to Design for X.* PhD thesis, Technische Universiteit Delft, September 1997.

[51] Zhihui Yao. *Constraint Management for Engineering Design.* PhD thesis, Cambridge University Engineering Department, April 1996.