

A Language for Building Concurrent Engineering Design Advisors: An Extended Abstract

James Bowen and Dennis Bahler
Department of Computer Science
and

Laboratory for Intelligent Systems in Design and Manufacturing
North Carolina State University
Raleigh, NC 27695-8206

1 Introduction

Concurrent Engineering (also known as Simultaneous Engineering or Life-Cycle Engineering) is an approach to design which takes into account not just the functionality of a product but also its manufacturability, testability, maintainability, and so on [National Science Foundation, 1987]. We aim to develop a generic programming technology which can be used to produce on-line design advisors that will encourage human designers to adopt a life-cycle perspective. To that end, we are using constraint networks to represent in an explicit way the mutually constraining influences that are exerted by a product's functionality, the material from which it is made, and the various processes involved in its manufacture, testing and maintenance.

A constraint network is a collection of objects and a set of constraints that specify relationships which must be satisfied by the values that are assumed by the objects. A major attraction of these networks is that the constraints can support non-directional inference. This means that when values are acquired by any of the objects involved in a constraint, values can be inferred for other objects attached to the constraint. Thus, for example,

a constraint network can capture the effect that a design decision has on manufacturing options. Equally, the same network could limit the designer's options if manufacturing decisions are made early on.

Several researchers have investigated or proposed the use of constraints in design-related tasks, in different design domains (see [Bowen and O'Grady, 1989] and [Serrano and Gossard, 1988] for surveys). Several constraint programming languages have also been developed (see [Bowen and O'Grady, 1989] for a survey). While these languages introduced seminal ideas, each one has some deficiency or other which renders it unsuitable for our kind of application.

Our research is aimed at developing a generic constraint programming language for Concurrent Engineering applications. Our methodology has been to develop a series of prototype languages, based on a study of the nature of life-cycle information in a variety of application domains. To date, we have investigated four main application areas:

- plastic parts
- extruded metal parts
- printed wiring boards
- elevator shafts

We have used our language prototypes to implement life-cycle design advisors in these areas. Each design advisor contains knowledge about the generic life-cycle restrictions that apply to a specific class of product. Decisions made by the designer of a particular artifact within this class are monitored by the design advisor. Whenever any decision, or one of its consequences, contravenes a life-cycle constraint, the designer is informed. Experience acquired from building these design advisor applications was used to refine successive generations of the language.

Experience with further applications is needed before we can claim to understand the complete range of language features needed to support Concurrent Engineering applications, but our experience to date, which is reflected in Galileo, the current version of our constraint programming language,¹ indicates that the following features, at least, are needed:

¹The previous version of the language was called Leo [Bowen and O'Grady, 1989].

- the predefined object domains must include real and integer numbers and arbitrary scalars, as well as sets and sequences of these;
- facilities for defining, both extensionally and intensionally, new application-specific object domains must be provided;
- the predefined functions must include the complete range of arithmetic and trigonometric functions, as well as set- and sequence-based functions (\cup, \cap , cardinality, length, head, tail);
- facilities for defining, both extensionally and intensionally, new application-specific functions must be provided;
- the predefined constraint predicates must include equality, numeric inequality ($>, <, \leq, \geq$), as well as set- and sequence-based predicates ($\in, \supset, \supseteq, \subset, \subseteq$);
- facilities for defining, both extensionally and intensionally, new application-specific predicates must be provided;
- the usual logic connectives ($\wedge, \vee, \Rightarrow, \Leftrightarrow$) as well as the negation operator must be provided;
- universal and existential quantification must be supported;
- the run-time system must provide facilities for nonmonotonic reasoning, as well as monotonic reasoning and explanation.

2 An Example Application: Printed Wiring Board Holes

Consider, for example, the task of designing a printed wiring board (PWB). Below, we give part of a Galileo program for this task; the program fragment concentrates on the subtask of designing the holes in the board. To life-cycle engineer a PWB hole, the entire life-cycle of the hole, and of the board to which the hole belongs, should be taken into account. The small program given here considers the functionality of the hole, the available drill bits and the bit magazine size of the hole drilling machine; it also considers

the reliability and assemblability of the board. A larger program, which is not given here for space reasons, considers a varied range of additional issues, such as whether the application is subject to military or commercial standards, whether the board is multi-layered and if so what layers of the board are linked by the hole, what the pad tolerances are, and what methods are available for producing the board artwork.

These issues are relevant to life-cycle design for holes. But the larger program as a whole is concerned with board design rather than just hole design so it considers many other issues, such as testability (necessary test regimes, available test equipment, bed-of-nails sizing, subcircuit isolation for test purposes, and so on). Even so, the program does not cover all the issues that are relevant to the life-cycle of a board. Indeed, it is probably impossible to specify the full range of considerations that are relevant to the life-cycle of any reasonably complex artifact. All one can do is arbitrarily define some sphere of interest and seek to address that, bearing in mind that it will always be possible to extend any chosen sphere. In PWB design, for example, most definitions of the relevant life-cycle sphere would not include ecology, yet the various chemicals used in etching a board, and the means by which they are disposed of, are intimately connected with the environment.

Thus, the example program given here is not intended to provide a full life-cycle treatment of PWBs or even just of PWB holes. Instead, it is intended to illustrate the nature of the programming technology we are developing and how this technology can be used to address life-cycle issues.

2.1 Holes in Printed Wiring Boards

To serve its intended function, the diameter of a hole must be large enough to accommodate the component lead as well as any copper and solder plating. But, we must also consider how the component lead is to be inserted into the hole. That is, the **min calculated hole diameter**² must exceed the sum of the component lead diameter and the thickness of any copper and solder plating by the amount of insertion oversize required by the board assembly system. If functionality and assemblability were the only concerns, all holes on a board could have the same diameter - the diameter required to accommodate the largest component lead. But in Concurrent Engineering this

²Bold-face lettering is used for text that is actually part of the Galileo program.

is not good enough - a hole can be too big for the component lead passing through it.

The need for mechanical and electrical reliability of lead connections places an upper limit on hole size. The diameter of a hole must not exceed the `max standard hole diameter`; that is, the difference between the hole diameter and the nominal component lead diameter must not exceed the `standard oversize`, which depends on the shape of the component lead and whether or not the hole is plated through. The need for reliability also places an additional lower bound on hole diameter, different from the one based on functionality and assemblability given in the previous paragraph. To allow solder to flow freely within a hole, and thereby achieve a good connection, the diameter of the hole must also be at least as large as the `min standard hole diameter`, which depends on the `class of the PWB application` and the `board thickness`.

If these functionality, assemblability and reliability factors were the only ones that needed to be considered, we could end up with boards having as many different hole diameters as component types. However, in Concurrent Engineering we also need to consider the economics of manufacturing the board. We must ensure that only drill bits of standard sizes are needed to drill the holes. Furthermore, we need to ensure that the total number of different drill bits needed to manufacture a board does not exceed the capacity of the drilling machine's bit magazine.

2.2 A Galileo program for PWB hole design

The Galileo program fragment which implements the requirements outlined in the last section illustrates usage of many of the features we mentioned earlier as being necessary in a language for building design advisors for concurrent engineering.

The object declarations are as follows:

```
/*OBJECT DECLARATIONS*/  
'the required insertion oversize'(ovrsz) : positive number.  
'the min standard hole diameter'(mnszd) : positive number.  
'the fractional factor'(factr) : factor.  
'the class of the PWB appln'(class) : appln_class.  
'the size of the drill magazine'(magsz) : positive integer.
```

```
'the board thickness'(brdtk) : positive number.
'the number of chosen hole sizes'(noszs) : nonnegative integer.
'the list of standard drill sizes'(lstdsz) : set of positive number.
'the list of chosen hole sizes'(lchsz) : set of positive number.
```

Each object is declared to have a short name, a longer synonym (which is used for the user interface) and a domain or type.

Most of the objects in this program belong to predefined basic domains, but some belong to application-specific domains, which are defined as follows, where the symbol ::= can be read as “is defined to be”.

```
/*DOMAIN DEFINITIONS*/
domain hole_type ::= {unsupported,plated_thru}.
domain oversize ::= {0.020,0.028}.
domain factor ::= {2/3,1/2,1/3,1/4,1/5}.
domain appln_class ::= {X : integer(X) and 1 =< X =< 5}.
domain lead_shape ::= {round,flat}.
domain hole ::=
    ('the hole type'(hotyp) : hole_type,
     'the nominal lead diameter'(nmldd) : positive number,
     'the plating thickness'(platk) : nonnegative number,
     'the solder plating thickness'(spltk) : nonnegative number,
     'the copper plating thickness'(cpltk) : nonnegative number,
     'the standard oversize'(stovr) : oversize,
     'the lead shape'(ldtyp) : lead_shape,
     'the max standard hole diameter'(mxshd) : positive number,
     'the min calculated hole diameter'(mnchd) : positive number,
     'the selected drill size'(sedsz) : positive number).
```

A constraint is a sentence in logic, consisting of a predicate applied to a group of arguments. The constraint declarations for this application are as follows:

```
/*CONSTRAINT DECLARATIONS*/
noszs =< magsz.
noszs = card(lchsz).
mnshd = factr*brdtk.
all X : hole(X) implies
```

```

(X.mnchd = X.nmldd+2*X.platk+ovrsz.
 compatible(X.hotyp,class,factr).
 mnszd =< X.sedsz =< X.mxshd.
 X.hotyp = plated_thru implies X.stovr = 0.028.
 X.hotyp = unsupported and X.ldtyp = flat
             implies X.stovr = 0.028.
 X.hotyp = unsupported and X.ldtyp = round
             implies X.stovr = 0.020.
 X.platk = 0 equiv X.hotyp = unsupported.
 X.sedsz in !lchsz.
 X.platk = X.cpltk+X.spltk.
 X.mxshd = X.nmldd+X.stovr.
 X.sedsz in ldsz.
 mnchd =< X.sedsz).

```

Most of the constraints in this program use predefined predicates, but one of them uses an application-specific predicate, which is defined as follows:

```

/*RELATION DEFINITION*/
relation compatible(hole_type,appln_class,factor) ::=
  {(unsupported, 1, 2/3),
   (unsupported, 2, 1/2),
   (unsupported, 3, 1/3),
   (unsupported, 4, 1/4),
   (unsupported, 5, 1/5),
   (plated_thru, 1, 1/3),
   (plated_thru, 2, 1/3),
   (plated_thru, 3, 1/4),
   (plated_thru, 4, 1/5),
   (plated_thru, 5, 1/5)}.

```

Two points need emphasis here. First, any syntactically correct 1st order logic sentence is an acceptable constraint in Galileo. Second, the user (the designer of a particular artifact) can introduce new objects and constraints as he interacts with a design advisor implemented in Galileo. Notice that in the program fragment given here there is no hole object even though there is a constraint which is universally quantified over all holes. Whenever the designer adds a new hole to his design (i.e., adds a new hole object to

the constraint network) this new hole will become subject to the universally quantified constraint which appears in the design advisor.

References

- [1] Bowen, J. and O'Grady, P., 1989. "A Constraint Programming Language for Life-Cycle Engineering", Report TR-89-01, Laboratory for Intelligent Systems in Design and Manufacturing, North Carolina State University.
- [2] National Science Foundation, 1987. "Research Priorities for Proposed NSF Strategic Manufacturing Research Initiative".
- [3] Serrano, D. and Gossard, D., 1988. "Constraint Management in MCAE", in J. Gero (ed.), *Artificial Intelligence in Engineering: Design*, Computational Mechanics Publications, 217-240.