SUPPORTING MULTIPLE PERSPECTIVES

A constraint-based approach to concurrent engineering

J. BOWEN and D. BAHLER
Department of Computer Science
North Carolina State University
Raleigh NC 27695-8206 USA

Abstract. In on-line systems intended to support Concurrent Engineering it is necessary to support the multiple perspectives that are taken by members of the various engineering disciplines involved. We report on Galileo2, a constraint programming language that enables constraint networks to be divided into different (possibly overlapping) regions, called fields of view. Engineers from different disciplines, who have different perspectives on the product development process, are assigned different fields of view when they interact with a program written in the language. Different fields of view may be presented through different types of interface, as dictated by the type of user who is expected for each field of view. Users of different fields of view may be given different levels of permission to edit the constraint network which they all share. The run-time system for the language facilitates interaction between users of different perspectives in the event of incompatible decisions by these users.

1. Introduction

Concurrent 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. One way to do Concurrent Engineering is to use design teams, comprising professionals interested in each phase of the product life-cycle. Throughout the process of generating the design, the designer receives comments from the other members of the team (test engineers, manufacturing engineers, etc.) on his evolving design. However, although some companies are starting to use them, design teams present many logistic, scheduling and other management difficulties.

Consequently, we are developing a programming language to enable the easy construction of on-line design advisors which would help ensure that designed products perform as well as possible in all phases of their life cycle. There are four key requirements for such a technology. It should:

- facilitate the representation of such entities as the artifact being designed, the components from which the artifact is configured and the environment in which the product will be manufactured, tested and deployed;
- facilitate explicit representation of the mutually constraining influences that these entities exert on each other;
- facilitate the construction of application programs which can take account of the differing interests of various members of a product development team, by presenting each member with an interface appropriate to his needs; and
- support interaction between these users when their decisions conflict.

J. S. Gero (ed.), Artificial Intelligence in Design '92, 85–96.

Research to date (Bowen and Bahler, 1991a, 1991b, 1992; Bowen, Bahler and Dholakia, 1992) has indicated that frame-based constraint networks are a suitable basis on which to build a language for Concurrent Engineering applications. A *constraint* is a declarative statement which specifies some requirement that must be satisfied by the values assumed by a set of related parameters. A constraint *network* (Mackworth, 1987) is a set of such constraints which are interconnected by virtue of sharing parameters. A *frame-based* constraint network is a constraint network in which parameters need not be scalars; specifically, in these networks, parameters can also be frames (Minsky, 1975) that is, data structures which are organized in an inheritance hierarchy.

In a frame-based constraint network, frames can be used to represent: the artifact being designed; the components from which the artifact is configured or the materials from which it is made; and the life-cycle environment in which the artifact will be manufactured, tested and deployed. Constraints can be used to express in an explicit way the mutual restrictions exerted on each other by artifact functionality, component/material properties, and life-cycle processes.

A major attraction of constraint networks for Concurrent Engineering is that constraints support multi-directional inference: information can flow in any direction through a network. Thus, for example, the impact of a design decision on the options available to a test engineer can be determined by propagating the design decision and its consequences throughout the network. Equally, if testing decisions are made early on, the impact of these decisions can be reflected in restrictions placed on the designer's freedom. By supporting multi-directional inference, one constraint network can support both of these forms of interaction with equal ease.

Several researchers have investigated or proposed the use of constraints in design-related tasks, in different application areas, including architecture, electrical and electronic circuits and systems, experiment planning, job shop scheduling, process planning, mechanical design, road design, and space planning. We provided an extensive application bibliography in (Bowen, O'Grady and Smith, 1990), and there is an excellent review in (Serrano and Gossard, 1988).

Several constraint programming languages have been developed; we provided a review in (Bowen, O'Grady and Smith, 1990). However, most of these languages, for example (Sutherland, 1963; Borning, 1981; Gosling, 1983), were developed for specific application areas, such as graphics and simulation, which are quite different from Concurrent Engineering; as a result, these systems are not appropriate for our area of interest. Even systems developed for design applications, such as CONSTRAINTS (Sussman and Steele, 1980) which was used for analysis and synthesis of electronics, or the Concept Modeler (Serrano and Gossard, 1988) which was used for Mechanical Engineering applications, do not offer the required facilities. Nor do languages which attempt to be general purpose, for example CLP($\Re$), CHIP or Prolog III (Heintze, Michaylov and Stuckey, 1986; Graf, Van Hentenryck, Pradelles and Zimmer, 1989; Colmerauer, 1987).

This situation motivated us to develop a generic constraint-based programming

language for Concurrent Engineering applications. In this paper, we report on certain aspects of Galileo2, the latest version of our language. In Galileo2, frames are used to represent artifacts, components, materials and life-cycle environments, while constraints are used to explicitly represent the mutual restrictions that these entities exert on each other. In this paper, however, we focus on how Galileo2 supports the multiple perspectives that are held by the different members of a product development team.

There are three aspects of the language which support multiple perspectives. First, Galileo2 enables constraint networks to be divided into different (but possibly overlapping) regions, or fields of view, each of which contains only the parameters that are of interest to the holder of one type of perspective. Second, the language allows these different fields of view on the same network to be presented through different styles of interface, including spreadsheets and feature-based CAD. Thus, for example, if one parameter belongs in two different fields of view, it may be shown as a graphical icon in one field of view and be presented as a spreadsheet cell in the other. Third, in the event of conflicting decisions by users from different perspectives, the run-time system for Galileo2 encourages interaction between these users in order to overcome the conflict.

In this paper, we introduce Galileo2 by presenting extracts from an example scenario which shows some interactions between a Galileo2 application program for printed wiring board (PWB) design and various members of a Concurrent Engineering team.

In Section 2, we introduce some aspects of the PWB application that have been chosen to form the basis of the example scenario. In Section 3, we briefly discuss the architecture of application programs in Galileo2 and illustrate our remarks with examples from the PWB application program, thereby setting the scene for the example scenario. In Section 4, we present extracts from the example scenario. In Section 5, we make some concluding remarks.

2. Aspects of the Example Application

To illustrate the language and the way in which it is intended to support the multiple perspectives held by different members of a Concurrent Engineering design team, we will consider some extracts from a series of interactions with a Galileo2 program that assists in the Concurrent Engineering of printed wiring boards.

In small companies, this program, which is called KLAUS, could function as a "virtual product design team," by monitoring the decisions made by a designer, as they are being made, and interjecting the kinds of comments and suggestions that would have been made by the members of a design team if the company had been able to establish such teams. In larger companies, where product design teams have been established, a distributed version of KLAUS could function as a clearing house for decisions made by members of a product design team who are unable to meet very often, perhaps because they have conflicting schedules or because they

are located in different plants.

KLAUS considers many aspects of the functionality, manufacturability, reliability, testability and cost of the board being designed. However, we restrict the example interactions that we present below to some extracts that consider issues of testability, as follows. It is often appropriate to introduce into a circuit components that are not needed for product functionality, but which are instead provided for testing purposes. For example, if a board contains a crystal that oscillates at a rate faster than that which can be accommodated by the available bed-of-nails tester, then an ancillary divider circuit for the crystal should be designed into the board. This means that, from the test engineer's perspective, the salient features of the life-cycle environment include attributes of the available test equipment, most notably in this scenario the maximum frequency testable by that equipment.

3. Programs in Galileo2

A program in Galileo2 comprises a set of declarations and definitions, which can appear in any order. A Galileo2 program is interactive, user input being in the form of additional declarations. Users can input their declarations in any order and a user can withdraw, at any time, any declaration that he has made in the past. The program specifies an initial constraint network, while user inputs extend this network.

In a program, the declaration statements include declarations of the parameters that exist in the initial network, the constraints that exist between these parameters, and which types of parameter, if any, users may add to the network at run-time. (There is no restriction on the type of constraint that may be added by users at run-time.) The definition statements in a program include definitions for the application-specific domains, functions and predicates that are used in parameter or constraint declarations, as well as boundaries for the various subnetworks (called "fields of view") that may be seen by different members of a design team. By default, a field of view is presented to its users through an interface based on a single-column spread-sheet or "scrollsheet." However, if any field of view is to be presented through a feature-based CAD interface, the definition statements in a program must include definitions for the icons that will be used to represent the parameters in that field of view.

In KLAUS, the parameters in the network represent salient attributes of the board, the circuit on the board, the components of this circuit and the environment in which the board will be manufactured and tested. The constraints in the network represent the mutual restrictions that the board, the circuit, the components and the environment exert on each other. The region of the network that is seen by a KLAUS user (i.e., his field of view) depends on whether he is a test engineer, circuit designer or manufacturing engineer; a field of view is defined for each such class of user. The test and manufacturing engineers see their fields of view through a scrollsheet interface while the circuit designer is given a CAD interface. Each

user can modify only that part of the network that is within his field of view. Since parameters in one field of view are connected by constraints with parameters in the other fields of view, decisions made by users flow across the boundaries between fields of view.

3.1. Building a Network

In Galileo2, a parameter declaration names the parameter, specifies the corresponding domain of legal values and provides an optional long synonym. For example, the following declaration from KLAUS

```
'the facility where the board will be tested'
(the_test_facility) : test_facility.
```

introduces a parameter called `the_test_facility`, assigns to it the long synonym "`the facility where the board will be tested`" (which is how the parameter will be named in all interactions with users), and specifies that the parameter takes its values from the domain `test_facility`. Long synonyms, which are arbitrary strings delimited by apostrophes, enable the Galileo2 run-time system to construct meaningul natural language utterances whenever it is outputting textual information to users.

If the set of domains that are predefined in Galileo2 are not adequate for a program, application-specific domains can be defined. Consider, for example the domain `test_facility` from which the parameter `the_test_facility` declared above takes its values. This domain could be defined as follows:

```
domain 'description of the available test facility' (test_facility)
=::=
('the name of the equipment at' (name) : tester_name,
 'the maximum clock frequency testable at' (maxfreq) : frequency,
 'the maximum number of test points testable at' (maxtpoints)
```

Any well-formed sentence in first-order logic is a well-formed constraint in Galileo2, including atomic, compound and quantified sentences. These sentences can use either predefined or application-specific predicate and function symbols. See (Bowen and Bahler, 1992) for details about how Galileo2 supports the definition of application-specific predicates and functions.

In writing a Galileo2 program, constraints are written using the short names for the parameters, predicates and functions involved but, if these symbols have been given long synonyms, the long names are used in explanations to the user. For example, given the long synonyms declared in the program for `the_test_facility` (see above) and other parameters, as well as the long synonyms given in the definitions for the `test_facility` (see above) and other domains, the constraint

```
the_test_facility.maxtpoints >= the_circuit.totaltstpts.
```

would be paraphrased by the Galileo2 run-time as follows:

```
the maximum number of test points testable at
the facility where the board will be tested >=
the total number of test points in the
circuit being designed.
```

3.2. Partitioning the Network

A field of view definition specifies the subset of the network parameters that may be seen within that field of view. Several fields of view are defined in KLAUS, one for each type of user. The following statement, for example, specifies that in the field of view called `testability` (which is the field of view that will be seen by test engineers) there is only one parameter, namely the one called `the_test_facility` whose definition we saw earlier:

```
field testability =::= {the_test_facility}.
```

On the other hand, the definition

```
field configuration =::= {X : component(X)}.
```

specifies that in the `configuration` field of view (which is the one seen by the circuit designer) all, and only, the parameters of domain `component` will be seen.

The number of parameters present in a network can increase at run-time. Users can declare new parameters and, as we we will see in the example scenario, new parameters can also be created because the necessity of their existence is inferred by the run-time system.

We can specify which users are allowed to declare new parameters and can specify what types of parameter they are allowed to declare. For example, in the KLAUS application, the statement

```
permission({X:component(X)}, configuration).
```

specifies that the user of the `configuration` field of view, i.e., the circuit designer, can add new parameters of domain `component`, to represent components that he decides to add to the circuit.

There is no restriction on the types of constraint that may be input by a user at run-time, except that each constraint must reference at least one parameter within his own field of view. That is, a user can specify a constraint between a parameter in his own field of view and parameters which may exist in other fields of view. But he must mind his own business, at least to the extent that he may not specify a constraint which references only parameters that "belong" to other users.

Galileo2 support three kinds of interface: feature-based CAD interfaces, scrollsheet interfaces and topological interfaces. Different fields of view in a network can be presented to users through different types of interface. KLAUS provides a feature-based CAD interface to the `configuration` field of view which is seen by circuit designers. It provides a scrollsheet interface to the other fields of view, which are seen by test and manufacturing engineers. Scrollsheet interfaces are provided automatically by the run-time system and require no programming. The icon-related definitions and declarations necessary to provide a feature-based CAD interface include: icon paintings; mappings between the icons and the network parameters they represent; and, if icons must change appearance when parameters change state, specification for how icon appearances and parameter states are related. Although our example scenario will include interactions with a feature-based CAD interface, the necessary icon-related definitions and declarations are beyond the scope of this paper and will not be considered.

4. An Example Scenario

A life-cycle advisor written in Galileo2 can be used for multiple design projects. To use the advisor on a particular project, we activate the advisor and then do a database read to load the network state with data specific to the project. To move on to another project, we save the current network state to the database, clear the network state, and load data on the next project from the database.

In the following scenario, we assume that a project leader has set up a database entry for a new project. We take up the story when the test engineer, who in this case is the first team member to make some decisions about this new project, starts to interact with KLAUS about the project. (We emphasize, however, that this scenario is only one of many that are possible: users from different perspectives can interact with KLAUS in any order desired.) Fig. 1 shows the interface presented by KLAUS to the test engineer after he has selected the test equipment to be used for the project. The largest window in this screen is a scrollsheet, in which each cell occupies one or more lines.

```
[ ]Help     [ ]File     [ ]New     [ ]Utilities     [ ]Search
[ ]Up       [ ]Down     [ ]Focus   [ ]Toggle
[ ]the name of the equipment at the facility where the      erdsys
    board will be tested
[ ]the maximum clock frequency testable at the facility     9.8
    where the board will be tested
[ ]the maximum number of test points testable at the        200
    facility where the board will be tested

>>>

KLAUS - a PWB Design Advisor (Testability)
```

Fig. 1

Fig. 2 shows the perspective of the circuit designer. Notice that this perspective provides a feature-based CAD interface rather than the scrollsheet interface provided to the test engineer. The drawing in this screen represents the circuit after the designer has introduced a CPU, and KLAUS has used its knowledge to infer the need for an associated pullup resistor and oscillating crystal. KLAUS introduced these components into the circuit and the designer has decided to accept these suggestions as part of his design.

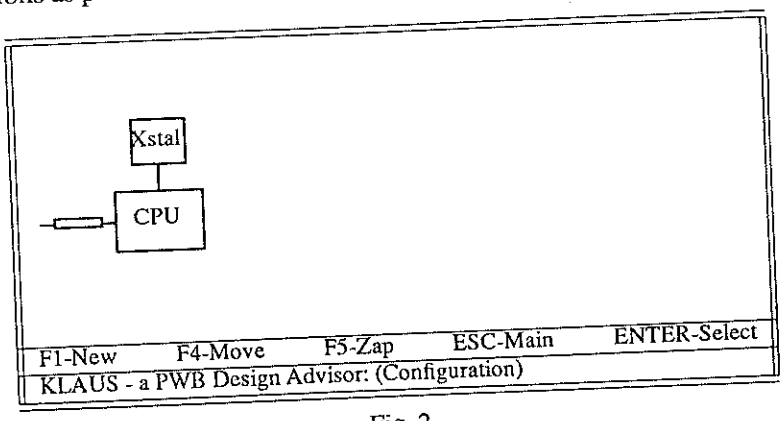

Fig. 2

Suppose that the designer, having accepted the crystal, specifies that it should oscillate at 25 MHz. Now, however, the following universally quantified constraint comes into play:

```
all X : osc_crystal(X) and the_test_facility.maxfreq < X.freq
implies
exists(X.'an ancillary divider circuit for'
        (anc_divider) : divider).
```

This universally quantified constraint specifies that every crystal oscillator must satisfy the following: if the maximum clock frequency testable at the facility where the board will be tested is less than the the oscillation frequency of the crystal oscillator, then an ancillary divider circuit for the crystal oscillator must exist. Although Galileo2 supports both universal and existential quantification, the symbol `exists` in this constraint is not an existential quantifier. It is a free logic existence specifier (Lambert and Van Fraassen, 1972), as explained in (Bowen and Bahler, 1991a), which discusses the theoretical basis for the dynamic introduction of new parameters into constraint networks at run-time.

Because of this constraint and because 25 MHz exceeds the maximum testable frequency of 9.8 MHz specified earlier by the test engineer (Fig. 1), the system introduces an ancillary divider circuit for this oscillator. The result can be seen in Fig. 3, where KLAUS is suggesting that the new component on the screen should be added to the circuit.

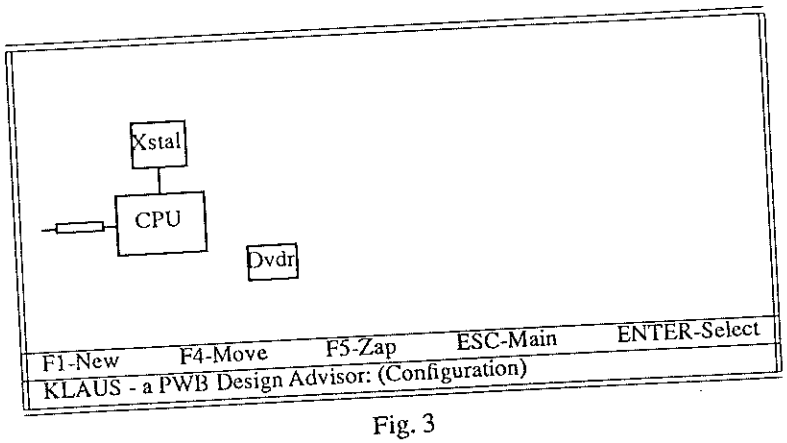

Fig. 3

The designer is surprised by the introduction of this component, so he asks KLAUS to justify it. In KLAUS's explanation (Fig. 4), the constraint given above, which introduced the component, is paraphrased in natural language; note the use in this paraphrase of the natural language synonym "`an ancillary divider circuit for`" for the parameter "`anc_divider`," which was defined in the constraint (see above) that inferred the necessity of this parameter's existence.

Despite this justification, the designer decides to reject this ancillary component. Now, however, the constraint which wanted to introduce the ancillary divider is violated, leading to the message shown in Fig. 5.

```
Justification
an ancillary divider circuit for the oscillator for the cpu exists
because of the following constraint
      (38) every crystal oscillator must satisfy the following:
                if the maximum clock frequency testable at
                the facility where the board will be tested <
                the oscillation frequency of the crystal oscillator
                then an ancillary divider circuit for the crystal oscillator
                must exist and must be a frequency divider;
and because of the following parameter value(s):
      the maximum clock frequency testable at the facility where
                the board will be tested = 9.8;
      the oscillation frequency of the oscillator for the cpu = 25.

the maximum clock frequency testable at the facility where the board will
      be tested was established according to the perspective taken by
      test engineers.

the oscillation frequency of the oscillator for the cpu = 25
because you said so.
```

Fig. 4

Choosing among the suggestions offered in this message, the designer decides to disable constraint (38). However, this constraint also refers to a parameter in

the testability field of view, so the decision to disable the constraint must be accepted by the test engineer. Whenever a user disables a constraint other than one he previously asserted himself, he is required to enter a free-text explanation of his action, which is saved for possible use in a design audit. These free-text explanations are also relayed by the system as a kind of electronic mail between users of different perspectives.

```
VIOLATION:
an ancillary divider circuit for the oscillator for the cpu
should exist, but is prohibited.

SUGGESTIONS:
(1) Retract the constraint
      (104) an ancillary divider circuit for the oscillator
                for the cpu should not exist.
(2) Disable the constraint
      (38) every crystal oscillator must satisfy the following:
                if the maximum clock frequency testable at
                the facility where the board will be tested <
                the oscillation frequency of the crystal oscillator
                then an ancillary divider circuit for the crystal oscillator
                must exist and must be a frequency divider.
(3) Change the oscillation frequency of the oscillator for the cpu.
(4) Request that, in the perspective taken by test engineers,
      a change be made to
                the maximum clock frequency testable at the facility where
                the board will be tested.
```

Fig. 5

When the test engineer next logs into KLAUS, he is told that a constraint of interest to him was disabled by another user. Checking on this, he calls up the design state from the database. The system tells him which constraint was disabled and produces the free-text explanation given by the circuit designer.

The test engineer decides that he is unwilling to allow this constraint to be disabled because of the difficulty in testing that would result. However, to compromise, he changes the test equipment to one which is able to handle a frequency of 25 MHz. After making this change, the test engineer reactivates the disabled constraint (38). Because of the higher frequency testable by the new piece of test equipment, the re-enabled constraint does not attempt to re-introduce an ancillary divider circuit, so no constraint violation occurs. The test engineer saves the new design state and starts to work on another project.

When the circuit designer next logs in, he is told that the test engineer has re-enabled the constraint but, since the unwanted divider circuit has not reappeared, the designer is content.

Suppose, however, that no such easy compromise was possible. For example, the designer might have selected an oscillation frequency that exceeded even the upper limit of the fastest available tester. In this case, the test engineer and designer

will successively disable and reenable their shared constraint (38), offering free-text explanations to each other until one or the other gives way or appeals to the project leader. In this case, the test engineer could give give way by deciding to build a special divider test fixture; the circuit designer could give way by using a lower oscillation frequency.

5. Concluding Discussion

In Concurrent Engineering advice systems, it is necessary to support the multiple perspectives on life-cycle design that are taken by members of the various engineering disciplines involved and to facilitate interaction between these parties when they make conflicting decisions.

In this paper, we have shown how Galileo2 enables constraint networks to be divided into different fields of view to reflect the different perspectives of various members of the Concurrent Engineering design team. We have shown that these different fields of view can be presented to users through different styles of interface, including feature-based CAD interfaces and scrollsheets. We have also shown that the run-time system for the language facilitates interaction between the various members of a product development team when they make conflicting decisions.

From the wide variety of experimental applications on which we have applied Galileo2, the language seems to offer a suitable environment for building Concurrent Engineering systems. However, it still remains to be validated in large-scale industrial applications. This process has now started, with the development of a Design For Solderability advisor which has been commissioned by a local manufacturer of printed wiring boards. Several theoretical questions also remain open. In particular, we are investigating the use of negotiation techniques from the field of distributed AI (Werkman, 1991).

References

Borning, A.: 1981, The programming language aspects of ThingLab, a constraint-oriented simulation laboratory, *ACM Transactions on Programming Languages and Systems*, 3(4), 353-387.

Bowen, J, O'Grady, P. and Smith, L.: 1990, A constraint programming language for life-cycle engineering, *International Journal of Artificial Intelligence in Engineering*, 5(4), 206-220.

Bowen, J. and Bahler, D.: 1991a, Conditional variable existence in generalized constraint networks, *Proceedings of AAAI-91, National Conference of the American Association for Artificial Intelligence*.

Bowen, J. and Bahler, D.: 1991b, Supporting cooperation between multiple perspectives in a constraint-based approach to concurrent engineering, *Journal of Design and Manufacturing*, 1, 89-105.

Bowen, J. and Bahler, D.: 1992, Frames, Quantification, perspectives and negotiation in constraint networks for life-cycle engineering, *International Journal of Artificial Intelligence in Engineering* (in press).

Bowen, J., Bahler, D. and Dholakia, A.: 1992, An AI constraint network-based approach to bed-of-nails DFT for digital circuit design, *Computers and Electrical Engineering* (in press).

Colmerauer, A.: 1987, *An Introduction to Prolog III*, Draft, Groupe Intelligence Artificielle, Universite Aix-Marseille II, November.

Gosling, J.: 1983, Algebraic constraints, *Technical Report CS-83-132*, Computer Science Department, Carnegie-Mellon University, Pittsburgh.

Graf, T., Van Hentenryck, P., Pradelles, C. and Zimmer, L.: 1989, Simulation of hybrid circuits in constraint logic programming, *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 72-77.

Heintze, N., Michaylov, S., and Stuckey, P.: 1986, CLP($\Re$) and some electrical engineering problems, *Proceedings of the 4th International Conference on Logic Programming*.

Lambert, K. and Van Fraassen, B.: 1972, *Derivation and Counterexample: An Introduction to Philosophical Logic*, Dickenson Publishing Company, Enrico, CA.

Mackworth, A.: 1987, Constraint satisfaction, *in* S. Shapiro (ed.), *Encyclopedia of Artificial Intelligence*, Wiley, New York, pp. 205-211.

Minsky, M.: 1975, A framework for representing knowledge, *in* P. Winston (ed), *The Psychology of Computer Vision*, McGraw-Hill, New York, pp. 211–277.

Serrano, D. and Gossard, D.: 1988, Constraint management in MCAE, *in* J. Gero (ed), *Artificial Intelligence in Engineering: Design*, Computational Mechanics Publications, Southampton, pp. 217-240.

Werkman, K.: 1991, Evaluating alternative connection designs through multiagent negotiation, *in* D. Sriram (ed.) *Computer Aided Cooperative Product Development*, Springer-Verlag, New York.

Sussman, G. and Steele, G.: 1980, Constraints—a language for expressing almost-hierarchical descriptions, *Artificial Intelligence*, **14**, 1-39.

Sutherland, I.: 1963, SKETCHPAD: a man-machine graphical communication system, *IFIPS Proceedings of the Spring Joint Computer Conference*, Spartan, Baltimore.