Solving Design Problems
by Computational Exploration

Dave Corne, Tim Smithers
and Peter Ross

DAI Research Paper No. 682

IFIP WG5.2 Workshop on Formal Design Methods for CAD, Tallinn, Estonia, 16-19 June , pp 249–270, 1993

Solving Design Problems by Computational Exploration

D. Corne$^{\mathcal{E}}$, T. Smithers$^{\mathcal{B}}$, and P. Ross$^{\mathcal{E}}$

$^{\mathcal{E}}$Department of Artificial Intelligence, University of Edinburgh, Edinburgh, UK.

$^{\mathcal{B}}$Artificial Intelligence Laboratory, Vrije Universiteit Brussel, Brussels, Belgium.

Abstract

Most real-world problems, especially design problems, are ill-structured, but formal approaches to problem-solving in AI have only really made progress into techniques for solving well-structured problems. Nevertheless, such research contains clues which illuminate the way towards formal approaches to solving ill-structured problems. This paper presents the foundations of an approach towards developing a better computational understanding of ill-structured problems and how to solve them computationally, with the eventual aim of giving AI programs a much greater and more useful role in the design process. The main issues which come up in this endeavour are the notions of different kinds of ill-structuredness, and the meaning of a 'solution' to an ill-structured (and hence possibly insoluble) problem. Some basic algorithmic recipes are proposed for dealing with the main kinds of ill-structuredness, and the initial design of a general computational technique which deals with general ill-structuredness is discussed.

1 Introduction

1.1 Overview

AI research has led to several kinds of problem-solving system. There is still, however, a long and complex process between a user's *identification* of a problem, and the eventual presentation of an abstracted, changed and generally 'treated' version of it for input to some AI program. Design problems in particular, with the current state of our problem-solving knowledge, usually need to be virtually solved before a version or variant of the problem is ready for presentation to a computer. With CAD, or Design Support Systems (DSS), we are able to use computational help at earlier stages of the process, but as yet chiefly to solve well-structured subproblems gleaned from the main design problem; the human user of such a system is still in the driving seat with respect to the long and complicated process of structuring, restructuring and incrementally revising the problem. Much current AI work can be seen as dealing with these difficulties, by investigating techniques which will enable programs to be consulted earlier in the design process. Two main sides to this are: enabling programs to *understand* the problem at an earlier stage, and: enabling programs to *solve* the problem *from* an earlier stage, while it is still typically an *ill-structured problem* (ISP). The former can be seen as being tackled by Natural Language research, for example, while the latter is the subject of this paper. That is, rather than the *presentation* of the problem, we are interested in ways to obviate the need to *structure* the problem, with the aim of developing the groundwork for AI theory and technology which will enable systems to deal effectively with inconsistent, incomplete, and/or imprecisely stated problems. Our motivation springs from the current bareness of AI's cupboard in the area of computational mechanisms for addressing ISPs, which is unfortunate in the light of the commonness of ISPs in the everyday world. Better understanding of ISPs will be very important for industry, and contribute to our understanding of everyday problem-solving.

The research programme outline in this paper concerns methods for finding solutions to ill-structured problems in general, and design problems in particular. The context is hence the general area of AI-in-Design, as well as various areas of problem solving research in AI. As we shall see, the context also includes research in cognitive modelling, and so setting the scene might unfortunately take several pages. Our basic underlying thesis, however, can be expressed with brevity, and is as follows:

Computational exploration can reliably determine *effective solutions* for an *important range* of *ill-structured problems.*

The rest of this introduction provides an elaboration of what is meant by *design problem*, discussed in the general context of ill-structured problems (ISPs), and outlines a method for investigating how to better understand them computationally, ending with some further discussion of what we currently seem to know about solving ISPs; In §**3.1** we elaborate on the particular background of this research: the work, since 1984, of the Design Research Group (DRG) at EDAI[1], which leads in to an outline of our 'exploration-based model of design'. Following a look at what it means to solve an ill-structured (and hence possibly unsolvable!) problem in §**4.1**, the remaining parts of §**4** set out ideas concerning investigating and understanding them via the presentation of some basic algorithmic recipes for handling three major kinds of ISP. Then, in §**4.5** we discuss the design of a general ISP-solver, based on the identification of what we think are a minimal and sufficient set of computational mechanisms which should lead to a high degree of effective ISP-solving, and hence design problem-solving. We end the paper with a brief summary in §**5**.

1.2 Ill-structured problems and design problems

A ubiquitous kind of intelligent behaviour is that which occurs when someone is faced with an *ill-structured problem* (ISP). These kinds of problems (see [Simon, 73]) can be ill-structured in three major senses ([Smithers & Troxell, 90]):

Inconsistency : The problem, as stated, cannot be solved!

Incompleteness : Not enough information is at hand to be able to solve the problem.

Imprecision : The problem has too many possible, different solutions.

One fairly evident thing about the definition of an ISP is that whether a problem is ill-structured or not depends on what the agent whose job it is to solve it knows. Another way of putting this is that it depends on the capability of the *process* which solves it, which can be viewed as the knowledge the process has access to and the knowledge-obtaining (*ie*: learning) mechanisms it has for modifying its knowledge. These are interesting and important points which we will get to later in §**4**. For now, it is interesting to note that most typical real-world problems are Ill-structured, in the sense of having one or more of the above properties. Strangely enough, when

[1] University of Edinburgh Department of Artificial Intelligence.

a problem is *inconsistent*, this doesn't stop people finding solutions to it. We will discuss what this means further in §**4**; basically, though, the solver does this by giving up on the problem as stated, *changing* the problem in some way (this does not mean that the problem is transformed to a new representation; rather, a *different* problem is devised), and then trying to solve the new problem. When a problem is incomplete, the typical strategy is for the agent trying to solve it is to use one or more means of finding the missing knowledge that will help solve it. This can involve learning more about the problem itself (gaining some generalised rules, or heuristics, from trial and error), basically via manipulating and exploring with the knowledge it has, or it can involve the agent going to a library, doing a PhD, learning a new language, or asking someone. The former way of dealing with incompleteness is the kind we plan to concentrate on here. The latter way of dealing with incompleteness which would require devising mobile robots which go to the library and read books; We do not propose to do this yet, but this would be an interesting and challenging extension to our work! Finally, when a problem is imprecise, this means that a point is arrived at where there are a wide range of possible solutions which, in themselves, differ in important ways (eg: a Ferrari and a Sinclair C5 are both possible solutions to "design a car"), but where the original problem statement does not help the solver in discriminating between them. The strategy here is for the solver to introduce new constraints and or objectives to augment those already present, which help discriminate between solutions in terms of important domain requirements which aren't expressed in the original problem.

Design problems are an interesting and widely tackled (by people) kind of ill-structured problem. Apart from a design problem's typical ISPness, what is generally mooted as being the characteristic feature of a design problem is that it involves transforming *performance requirements* (and/or *functional requirements*) for some particular kind of object (this constitutes the initial statement of the design problem) into *structural specifications* which determine exactly how to construct an object of the given type so that it meets the performance requirements. Awkwardly enough, it is hard to think of an ISP which *cannot* be put in these kinds of terms, but study of design problems in AI and elsewhere almost invariably deals with physical artefacts (eg: cars, buildings, photocopiers, rather than plans, natural language utterances, or strategies). Hence, design problems can normally be understood as the kind of problem that the design team in a manufacturing or engineering firm are typically faced with, or the kind that architects are typically faced with.

As with most real-world problems, design problems are typically ill-structured in the ways described above. If they weren't, design would be no problem. In §**2** we will begin to examine what AI-in-Design research has so far taught us about addressing ISPs. This is not very much as yet, since much AI-in-Design research has concentrated on the 'transformation of performance requirements to structural specifications' aspect of design problem solving, rather than ill-structuredness, as well. Although lots of *representation-oriented* work in AI-in-Design (*eg*: [Tham & Gero, 92] [Qian & Gero, 92]) will in future contribute greatly to this endeavour when, for example, design prototypes are used to support ISP-oriented problem solving processes.

The course of our reasearch into ill-structured design problems and formal methods for their solution falls into three planned stages:

1. Identify useful ways in which to understand and characterise different degrees and kinds of ill-structuredness (ie: devise a taxonomy of problem-types based on ill-structuredness, which the inconsistent/ imprecise/ incomplete triage represents the beginnings of).

2. Design, implement and investigate computational processes which address certain of these different kinds of ill-structuredness,

3. Design, implement and investigate *computational exploration*; a process which should be capable of taking a general ill-structured problem as input, and producing an effective

solution to it.

The important aspects of the work will be how successful we are at characterising different kinds of ISPness, in such a way that, for example, we are able to specify a particular solution-process for a non-trivial kind of inconsistency. A significant part of the work will take the form of trying to *unify* ideas and observations from diverse parts of the AI-in-Design and problem-solving literature which have a bearing on dealing with ISPness (but are pehaps not expressed in such terms). For example, much machine learning work can be seen as concerning itself with a certain kind of problem-incompleteness; a classifier system, for example, uses a genetic operators to modify its rule-base in order to try to discover useful rules for dealing with new situations. Also, Lenat's AM can be seen as embodying techniques to deal with imprecision; 'problems' input to such discovery systems are necessarily highly underconstrained.

The main theoretical and practical contributions will be the capture of general models for dealing with different kinds of ill-structuredness, and deriving, with the help of these models, the principled design of a *computational exploration* process which deals with a non-trivial range of combined types of ill-structuredness. This should help us get to grips with how the general design process can be formally modelled, and should lead to novel architectures for new kinds of automated design systems and design support systems. In the longer term, such ISP-solvers will be an excitingly novel kind of problem-solving system; probably less *correct* and/or *efficient* than typical WSP-solvers, but capable of being employed at much earlier stages in the real-world problem-solving process than current systems, which necessitates their being more individual, idiosyncratic, and non-deterministic. Throughout, and especially in these earlier stages, we welcome review, comment, ideas and perspectives on our work from members of the AI/Design community.

2 AI-in-Design

2.1 Automated Design

Here the emphasis is on buidling systems, often 'expert systems' in the classical sense, which are supposed to produce one or more designs for an artefact, given as input a set of specifications for that artefact. Some major centres if such research are the Rutgers AI/Design project ([Tong, 90]), the knowledge-based systems Lab at MIT ([Sriram *et al*, 89]), and the Engineering Design Research Center at CMU ([Westerberg *et al*, 89]). A great deal of other work at other places is also along these lines. The principal results and lessons which we can learn from looking at this work are twofold (and perhaps predictable): first, automated design is a very difficult process, which rises quickly in complexity and declines in tractability with the size of the problem being addressed; second, classical search-based methods are successful in solving highly abstracted forms of well-structured real design problems. In almost all examples of such work that we have found, the system only deals with well-structured problems; *ie*, there is no provision, for example, for providing a solution to an inconsistent problem, other than for the system to fail. An exception is Navinhandra's early work on CYCLOPS ([Navinchandra, 91]), a system which exhaustively searches for a solution, and, if it doesn't find one, chooses a constraint or objective to relax and starts all over again. Much of this work can be seen as tackling imprecision, in a sense, in that the system might embody *implicit* extra constraints or defaults, which augment those constraints in the problem statements put to it. The way in which imprecision is dealt with is thus rather raw and inflexible. Also, we can look at some of this work in terms of having something to say about a basic form of incompleteness. For example, [Steinberg & Ling, 90] considers the different behaviour (mainly in terms of speed) of an automated design system as a function of different amounts of *a priori* heuristic knowledge. On the whole though, research so far in the automated design field has provided a lot of effective techniques for solving well-structured design problems, but has made only scant foray into dealing with ill-structuredness in those problems. An interesting, and perhaps telling indication of this is the fact that we have rarely seen the literature of such work provide examples of *designs* produced by a system. Even more rarely have we found an example in which a system provided two different designs, in different runs, for the same initial problem. A major consequence of ill-structuredness is that different designers will produce different results, or the same designer will produce different results at different times, but the lack of such considerations in automated design research seems to reveal that little attempt has yet been made to understand this process; to examine, for instance, how different *styles* of design might result from different process parameters. The only, and interesting, exceptions to this in the automated design field which we have found are Harold Cohen's work on AARON ([Boden, 90]), and Angela Wanken's MSc project on abstract graphic design ([Wanken, 91]). However, these are both at another extreme, in which the initial prohlem is *constrained* to be highly imprecise with regard to many details, and the program exploits this by making *random* choices from large sets of possibilities.

2.2 Design Support

Here the emphasis is on building systems which provide, to varying degrees, suitable, helpful, co-operative and non-prescriptive support for a human designer as he or she designs. A selection of centres and groups concerned with design support system rsearch includes the CAD-Centre at Strathclyde University, the Engineering Design Research Center at CMU, the Design Computing Unit at Sydney University, our own DRG at EDAI, and the ICADS group at CalPoly. The multiplicity of different kinds of design support being looked at is vast, but susceptible to generalisation into three major groups: work concerned with facilitating the *expression* of designs, work concerned with facilitating the *evaluation* of designs, and work concerned with facilitating the making of *design decisions* (ie: more directly than is the case in the other two groups). Facilitating *expression* is what is achieved quite well by a system comprising a pencil, a large loose-leaf pad, and an eraser; in design support research, these are replaced by a computer keyboard, a computer screen, and a delete key. The issues here are to provide facilities for storing, merging, keeping track of, structuring, and recording designs and partial designs, which are as flexible as possible (in the sense that the user can do as much of these things as he or she wants). Most such work, however, tells us little about the design process, or how we might computationally address ill-structured problems, except in the sense that it implicitly recognises the fact that the way people do this is extremely complex. The *evaluation* stream, on the other hand, concerns what to do with an expressed design; *eg.*, simulate its behaviour, analyse its performance, and so on. Again, while important in relation to the needs of industry, this tells us little about the design process itself except inasmuch as we can analyse how people actually use these systems. Broadly speaking, the *expression* and *evaluation* streams of research are the concern of the Computer Aided Design literature, rather than AI-in-Design, as such. The third group concerns more the use of AI-based techniques in the design support process. This group tends to be more interesting with regard to understanding design, since some degree of expression and evaluation is combined and used to support decision-making. For example, recent work at the EDAI DRG concerns making use of the user's dynamically changing input design decisions and requirements to guide evaluation towards what seems to be of most interest to the user at any particular time, as well as guide the system towards making relevant suggested design decisions. Other kinds of design support work concern a concentration on useful representations for design objects which are engineered to support a maximal degree of design oriented reasoning (*eg*: [Tham & Gero, 92]), as well as the use of machine learning techniques to infer knowledge for further design reasoning based on design exanples (*eg*: [Maher & Li, 92])

On the whole, then, design support research tends not to address too closely the matter of how the design process actually works. More often than not, the support system is a stand-in for other media for expression (eg: a drawing board), or other media for evaluation (eg: the real world, small-scale prototypes, or calculators), and the design process itself is still entirely (albeit implicitly) the user's responsibility. When sophisticated expression and evaluation considerations are somehow integrated, however, the potential is there to investigate in detail the overall control of the design process and important aspects of design reasoning, but results are as yet too fragmented and compartmentalised with respect to understanding the overall nature of ISP-solving. Design support systems, then, generally allow a user to pursue the business of ill-structured problem-solving, but they still don't shed much light on how they do it. Inconsistency is usually dealt with by the system helpfully noting an inconsistency, and the user taking any necessary action. Imprecision and incompleteness are usually matters entirely for the user to deal with him or herself.

2.3 Cognitive Modelling

Typical work in this theme has concerned itself with identifying production-rule based or blackboard-based models of simple design processes (eg: [Whitefield & Warren, 89]). Other work has concerned itself with analysing protocols of human designers, aiming to identify the central kinds of activity which occur in the process. A problem with this is that the results of a protocol are heavily influenced by what the researcher was looking for in the first place. For example, two characterisations of design behaviour from separate protocol studies in the same proceedings ([Visser, 91], [Colgan & Spence, 91]) posit almost mutually exclusive ideas about what these central activities are; on the one hand, [Visser, 91] champions *opportunism* as the essential characteristic of design behaviour, while [Colgan & Spence, 91] finds *hierarchical goal-oriented activity* as the best fit to protocol analyses. In two examples of more recent work, however, we can see progress being made. In [Davies & Simplicio-Filho, 92], a thorough study which combines theoretical and empirical research into software design, converging on the characterisation of design as an opportunistic + goal-oriented activity, in which strategies are continually being formed,but rarely fully followed. Also interesting is the work reported in [Goel & Pirolli, 92], in which extensive protocol analysis was used to characterise "invariant properties" of design problem spaces, which do not occur in more simple problems. The author includes much criticism of the failure of Simon's characterisation of 'design as search' ([Simon, 69]) in accounting for these aspects of design, in a similar vein to criticisms in [Smithers & Troxell, 90], and later in [Smithers, 92].

In cognitive modelling, then, simple design process models occur, just as they do in automated design research, but more recently we are seeing results concerning more complex and typical manifestations of design activity, in which clear differences are found between the kind of problem a typical design problem is and the kind of problem a well-structured problem is ([Goel & Pirolli, 92]), and in which we are seeing more clearly how people seem to address these problems ([Davies & Simplicio-Filho, 92]). It is interesting to note that inconsistency, incompleteness, and imprecision are included among the "invariants" found by Goel & Pirolli, and dealing with these aspects emerges as a result of intensive, incremental problem revision and 'exploration at tangents' activities found by Davies & Simplicio-Filho.

2.4 What we know about Design

We haven't yet learned very much about what we can computationally do, given a particular ill-structured problem. We know a lot, however, about how we can seem to help designers to design things, with AI techniques playing various different kinds of supporting roles. Boeing, for example, have recently designed their new passenger plane (the 757) entirely 'by computer'. All this means, however, is that they have made a great deal of use of CAD systems to help them design almost every detail of the plane's construction, to the extent that they haven't had to do much in the way of building prototype models of the plane. We also know a lot about how to solve various "peformance requirements $\Longrightarrow$ structural specifications" problems by well understood search and/or optimisation methods, and we are learning a lot about how best to represent design objects in order to support general design reasoning. But research is only beginning in AI into general computational techniques for addressing ISPs. In summary, there is a lot of work going on in AI-in-Design, but it is still early days in terms of being able to describe the sufficient computational mechanisms which will support general design behaviour. This has been addressed in Cognitive Science ([Goel & Pirolli, 92]), but researchers in AI have either not recognised it as a research issue, or kept putting it off until some time in the future.

One interesting reason for not recognising it as an issue, or not seeing it as worth investigating, is the idea that solving ISPs is so complex, involving so many different kinds of reasoning and being so knowledge-intensive, that all we can do is concentrate on small aspects of it — it is too difficult to understand it in the whole. The latter seems like a good reason not doing the research we are engaged in. However, there are several things wrong with such a position. To start with, this seems like a good argument for not studying machine learning, natural language, or almost any other subfield of AI that can be named. The only way to really find out whether something intricate can be understood is to have a go at investigating it; how this is done is largely the strategy of the second, and most telling, counter to such a position: this position confuses AI with psychology. We do not want to come up with a theory of how people design things, or a theory of how machines could get to design things like people do. All that we need in AI, with regard to ISPs, is a formal computational technique which can effectively solve design problems. There will be many kinds of such technique, or one general technique with many different variants, but at least one would be nice to start with. Generally, it would be interesting and important to know what are the *sufficient* computational processes required for solving problems with *particular* kinds of ill-structuredness. Accepting this, one might go on to say: "well, why not look around for existing AI-based models of different kinds of reasoning which you think are important for ISP-solving, and then work on ways of hybridising and/or getting co-operation between them". Well, this is a fine idea, which seems to have many precedents. The original research project of the EDAI design research group started with something like this, in the context of design support. Our work is indeed along such lines, but with a very important qualification: We would like to find *minimal* sets of computational mechanisms, which provide a *maximal* degree of effective design behaviour. This design behaviour should at least be effective in that it can handle ISPs with a certain degree of inconsistency, a certain degree of incompleteness, and a certain degree of imprecision. *Eg.*, at least as inconsistent as: "design a kitchen layout for space S which contains appliances A and has room for table T" (this may be unachievable, but certain changes *to the problem itself* may be made which could lead to a solution to this design problem), but not as inconsistent as: "find a decision procedure which can establish the truth value of any mathematical expression"; as incomplete as: "find a good next move for white in this game of 3D chess" for an agent who hasn't yet any experience of the game, but not as incomplete as "assess the legal ramifications of violating software patents" for an agent who has never studied law; and, as imprecise as: "design a Lisp program for adding six numbers" (there are countless ways to do this), but not as imprecise as "design a Lisp program".

To start having a go at this, the next section talks about the components of computational exploration: a technique which will be designed so as to be able to, hopefully, give satisactory answers to an important range of ill-structured problems. Before that, we will talk a little about the background to this paper; the work of the DRG at EDAI.

3 Computational Exploration

3.1 Background

The DRG at EDAI, now with some forty person-years of research effort behind it, has been concerned with both the theoretical and technical development of AI-in-Design, especially as concerns getting to grips with the notion of the ISPness of design problems ([Smithers *et al*, 90]). The long term aim of the research is to develop a computational theory of the design process, since such a theory is necessary for the principled development of computer-aided design tools and support systems, and their successful integration into industrial and commercial environments. It would also significantly advance our understanding of computational approaches to solving ill-structured problems. Funded AI in design research started in 1984 as part of the Alvey large scale demonstrator project *Design to Product* (DtoP). This work resulted in an innovative design support system architecture, known as the Edinburgh Designer System (EDS), which tightly integrates different types of knowledge representation, inferencing, and control systems in attempt to provide effective support for a designer. During the two research grants (GR/F/6723.5 and GR/F/001/0) which supported the DtoP work the group also began developing an *exploration-based model* of the design process, in order to model the group's incrementally develping theoretical understanding of the knowledge process that underlies design problem solving.

This exploration-based model attempts to identify the essential components required to adequately describe the design process, and does so by describing the design process as an ideal knowledge process at the *Knowledge Level*. Exploration starts with an *initial requirement description*, R_i which is interpreted in terms of knowledge of the domain, K_{dm} and knowledge of how to design in the domain, K_{dn}. Typically, R_i is ill-structured. In general it is not possible to identify in what way an R_i is incomplete, inconsistent, and imprecise to start off with, which is what makes the problem a design problem. The design process involves attempting to transform the initial requirement description into a final requirement description, R_f, which forms a complete, consistent, precise, and unambiguous description of the criteria that need to be satisfied by a design description, D_s, when realised, if it is in turn to satisfy the needs or desires that motivated the construction of the initial requirement description. This transformation process, which we call *exploration* involves the construction and incremental extension of problem statements and associated solutions. Sometimes this takes the form of simply constructing partial solutions which satisfy a subset of the requirements, but often this involves devising intermediate problems, at tangents to the main line of reasoning, designed so as to help elicit more knowledge about the possibilities in the domain. The key aspects of the process – requirements descriptions, intermediate problems constructed, partial solutions, and so on, are recorded in a design history H, which represents a kind of rationale for the exploration carried out. At any stage exploration is pursued by making changes to one or more of the current requirement descriptions, R_c, the current problem statement, P_c, or the current design description, D_c, and is continued until P_c forms a well-defined problem statement which embodies, via the constraints it identifies, all the criteria in R_c, and the solution to P_c specified by D_c satisfies R_c. The requirement description is then taken to be a final requirement description, R_f, and the design description is taken as the final design description, D_f. The complete history of the explorations engaged in to arrive at R_f and D_f is called H_f and represents another important component arising from the design process. Together, H, H_f, and K (which includes K_{dn} and K_{dm}), form the *Design Description Document*, (DDD).

Generally, we can describe this process as a collection of explorations each of which can be represented as a seven-tuple $[R, P, D, K, T, H, E]$, where:

- R is the, typically partially ordered, set $R_i, ..., R_c, ..., R_f$ of requirement descriptions generated as they are transformed from the initial description to a final description;
- P is the (partially ordered) set $P_1, ..., P_c, ..., P_f$ of incrementally constructed well defined problem descriptions formed by deriving constraints from some, and eventually all, of the criteria specified in R_c;
- D is the (partially ordered) set $D_1, ..., D_c, ..., D_f$ of design descriptions that form possible solutions to P;
- K is the domain and design knowledge available to support the exploration
- T is a set of functions that take elements of K, H, and P_c and return necessary and possible changes to the current design description, D_c which forms a solution to P_c;
- H is a record of what functions from T have been applied to which elements of K and P and the inferences produced; and
- E is a decision procedure which takes as input R_c, P_c, D_c, K, T, and H and outputs a set of changes or extensions to R_c, and/or P_c, and/or D_c, and any updates to H.

This understanding of the exploration process which underlies design problem solving is further discussed and explored in [Smithers, 92], and can be seen in earlier incarnations in [Smithers & Troxell, 90] and in [Smithers *et al*, 90]. This characterisation of exploration was arrived at from experience with trying to work out how to build supportive and collaborative design support systems, and watching them in action in conjunction with their human users. The characterisation is very wide-ranging, and, in essence, describes a highly complex and dynamical physical symbol system. This complexity and dynamicity can be seen as accounting for the fact that, given a particular designer (*ie.*, a particular instantiation of the design process) and a particular ISP, it is extremely hard, and possibly impossible, to predict what the resulting design will look like. The point is, however, that the exploration-based model is designed so as to at least have the wherewithal to make coming to a solution *possible*. That is, the exploration-based model ultimately intends to outline sufficient mechanisms for solving general ISPs.

3.2 Implementing Exploration

The research aim of a computational implementation of a subset of the exploration-based model comes from the desire to put to the test many of the ideas so far gained from our work with ideas in collaborative design support. Placing hese notions in the context of a fully-automated experimental design system will serve as a new wellspring of observations from which to further develop and refine the general formal model, which will hopefully then feed back into our main work of building increasingly supportive and collaborative design support systems. Ultimately this should also eventually lead to ideas for enhancing the capability of automated design systems and design support systems, and hence hopefully contribute a little to AI's currently small formal armoury for addressing ISPs. As noted in §**2**, the key to computational ISP-solving probably largely lies in the principled combination of some existing techniques (*eg.*, case-based reasoning + analogical reasoning + constraint-satisfaction + etc ....); one can therefore look at our research in terms of an attempt to find what the 'principled combination' might look like. Simon, for example, in [Simon, 69], appears to see multiple-problem-space search as the sole technique needed, and the principled organisation of these searches is a very clever, but completely unspecified, control regime. Interestingly, in attempt to put some flesh on such an approach, Steier ([Steier, 92]) describes ISP-solving as multiple-problem-space search, in which most of this clever control structure is implemented by the human user of such a system. However, our approach is, indeed, to take and perhaps modify some existing techniques, and combine them in a principled way. Our particular contribution comes in trying to establish the minimally sufficient set of such techniques that are necessary for solving an important subclass of ISPs.

We plan to do this by first looking into the different ways in which problems can be ill-structured, and then investigating general algorithmic recipes for dealing with these kinds of ill-structuredness in isolation. Via considering the latter at the level of the common subprocesses that occur in each class of algorithm, we will investigate principled ways to combine these algorithms so as to form an experimental, computational exploration system.

These ideas are described further later on, but right now it will be convenient to look at an important aspect of all this: how do you tell whether or not a solution to an ISP is an *effective* solution? That is, what must be true of a proposed solution to a design problem if we are to actually consider it as a solution? It is with this question that we can most clearly see the distinction between ISPs and well-structured problems (WSPs). In [Amarel, 90], for example, problem-solving research in AI is characterised solely in terms of problems where the only test for being a solution is that a solution *meets the original requirements*. This simply doesn't work for many ISPs; either it is impossible to meet the problem requirements, or there are simply too many possible solutions which do. In the latter kind of case, solving the problem will involve imposing more constraints, which will occasionally turn the problem into an inconsistent one. It is by no means unusual for a designer, in such a situation, to change the problem by violating an original problem requirement, rather than one of his or her imposed ones. Hence, solving ISPs typically results in a soluition which violates some, if not all, of the requirements in the original problem statement.

4 Solving ISPs

4.1 What is the solution of an ISP?

As soon as we escape from the bind of having to fully meet the requirements of the original problem statement, we find ourselves in a completely different world as far as judging the effectiveness of solutions is concerned. For, example, Amarel ([Amarel, 90]) goes to considerable length discussing problem-solving in AI, with a key unifying feature of such work being that in every case the solved problem must satisfy the requriements of the original problem statement. To step away from this, and start considering problems which are or become inconsistent, means that we have to add a completely new dimension to the theory of problem solving. This new dimension will have a lot to do with what we term *Problem Revision Paths* (PRPs). To motivate the need for and definition of a PRP, consider the following problem:

Design a crossword puzzle which has a square grid, with at least 95% of the cells containing letters of words, with all words being from the Concise Oxford English Dictionary, all words having four or more letters, and the grid pattern being symmetrical about its vertical and horizontal axes.

This problem is inconsistent, largely because of the "95%"requirement (typically, the fact that a problem is insoluble will be much harder to discover, if it can be discovered at all). A solution to this puzzle, then, will certainly not meet all of the requirements in the initial problem statement. The presence of such inconsistency means that we have to change the problem in order to solve it. Of course, we don't actually solve *it*, we solve a related problem. There are, however, very many possible ways in which we could change any problem; these are the various, essentially infinite number of ways in which we can alter, add, or remove requirements.

The 'different world' indicated above is precisely this world of possible *requirement revisions*. First, there is typically no indication in the original problem statement as to what kinds of revisions might be allowable, and secondly, no solution that we might come to from changing the problem in this kind of way is, in the 'old world' sense, a solution to the original problem. So, the *effectiveness* of a solution to a design problem solely in terms of the initial requirements simply doesn't make sense, even though this is the be-all and end-all as regards solution-testing for WSPs. On the other hand, there *must* be some way to judge the effectiveness of a design solution. We cannot go around changing problems willy-nilly, since this would allow us, conceivably, to change any design problem into "What is 6×7". The way we characterise what it means for a solution to an ISP to be effective comes from recognising the need that each requirement revision must be *justified* in some way. A solution to a design problem is therefore characterised by a solution to R_f, which can be judged in a normal sense (since R_f is always a WSP), coupled with a justified *Problem Revision Path* which links R_i to R_f. A typical arc in such a path would express something like "A lot of time T was spent in trying to successively incorporate more words, but well before 95% of the grid contained letters, it became unreasonably difficult to incorporate more words, hence it seems a good idea to relax the 95% constraint". The effectiveness of the solution itself comes down to the effectiveness of these individual justifications, and we can imagine ways in which this kind of effectivness can be specified. For example, we may only consider the relaxation of a constraint as justified if a proof has been found that the problem cannot be solved without this relaxation. This is quite a strict measure of effectiveness; in practice, it is interesting to note that what constitutes an effective requirement revision depends heavily on the judge of that solution (which may be the solver, an external judge, or a team of assessors), and constitutes a particularly interesting dimension along which different kinds of resulting designs will emerge.

The case is actually quite similar for imprecise problems. For imprecise problems, the design process must introduce new constraints in order to guide the process of solving them; not only must these new consraints be justified in some way, but their introduction might well lead to an intermediate problem in the sequence R_i ... R_f being inconsistent, even if R_i wasn't. The story for incompleteness is not quite the same however; dealing with incompleteness concerns mechanisms which discover new problem solving operators which augment the solution construction activities, as well as explorations which lead to the discovery of new constraints which augment the problem revision activities.

In judging a solution to a design problem then, we must consider the PRP, plus the final solution to the final revised problem. The PRP indicates why the problem was changed in the way it was. Every adjacent pair $[R_t, R_{t+1}]$ from R can be seen as having an associated requirement-revision arc, A_t, which details the reasons for undertaking that particular revision. The sequence $[A_i, \ldots A_{f-1}]$ will be a critically important structure in terms of the progress of the computational exploration process. If and when the exploration process finds a solution to a problem, this will mean, by definition, that it has found a sequence $A = [A_i, \ldots A_{f-1}]$, and a solution, S_f to the final set of requirements R_f. What sequence A implicitly says is "in terms of the bounds under which successive problem revisions were deemed *a priori* to be allowable, this represents an effective solution". The needs for a successful computational exploration process are therefore to incorporate requirements-revision bounds which disallow such excessive revisions as "I've spent five minutes on this crossword design problem and can't solve it, so I'll ignore all the constraints except for the fact that it has to be a crossword", but not restrict the revision process so far as to tenaciously hold on to R_x until either time or memory run out before a solution is found. An example of this kind of procedure can be seen in Navinchandra's CYCLOPS system ([Navinchandra, 91]), where the only possible kind of requirement-revision arc allowed is : "I have exhaustively searched for a solution to this problem and cannot find one; I will now randomly choose a constraint, and slightly relax it". This is what Navinchandra terms 'exploration'; we use this term differently, in such a way that this kind of behaviour turns out to be a special case, which is suitable only for small problems with certain kinds of constraints.

The core process which occurs in handling both inconsistency and imprecision is therefore *requirement revision*. For handling incompleteness, the core process is *knowledge revision*, or simply *learning*. Noting all this, we can begin to consider what it takes to solve inconsistent, incomplete, and imprecise problems in isolation (*ie.*, *only* inconsistent, *only* incomplete, and *only* imprecise problems). The next three subsections will consider such problems in turn, via constructing three simple, general classes of algorithm (not yet fully computational, and ignoring efficiency) which look like they might be sufficient for each of these jobs. We term these algorithms $\mathcal{I}1$, $\mathcal{I}2$, and $\mathcal{I}3$, and will later refer to them together as 'the $\mathcal{I}$-algorithms'. After presenting these $\mathcal{I}$-algorithms, §**4.5** will look at issues involved in implementing a general computational exploration process by in some way combining them. We will find, however, that rather more work has to be done than simply combining them, and we will start to consider in more detail the underlying processes which are common to one or more of the $\mathcal{I}$-algorithms.

4.2 An algorithm for inconsistent problems

Say we have a problem to solve, in the form of a set of requirements, R, and we have some domain knowledge K, which comprises knowledge of how to construct solutions, and some degree of heuristic knowledge abpout how to design things in terms of different kinds of requirements (ie: how to choose among solution construction operators). These requirements are basically *constraints*, and associated with any requirement is a space of *possible-changes*, $c(R)$, which enumerates a set of different versions of R. Eg: if R is a constraint which places parameter p in interval I, then $c(R)$ simply enumerates, or intensionally defines, the set of intervals which could replace I. A simple class of algorithms which address *inconsistent* problems is as follows::

Algorithm $\mathcal{I}1$:

1. Input: A problem R, resource parameters T, and knowledge K.
2. Apply resources T in order to solve problem R, yielding result S. Eg: apply `solve`(R,T,K), yielding S.
3. If S is a solution to R, then halt.
4. If S is not a solution to R, then apply `decide1`(S,R,T), yielding result Q.
5. If Q indicates that R is likely to be solved if more resources are applied (eg: if more time is spent on it), then return to 1, with a new resource bound (indicated by Q), and to continue from where 1. was interrupted.
6. If Q indicates that R is likely to be insoluble (with respect to the available resources, or insoluble in principle) then go to 7.

7. Execute `choose1`(R,K,T), to choose a constraint $C \in R$, and an element C_{new} of $changes(C)$ distinct from C, Replace C with this new constraint, thus forming a new problem R_{new}.

8. Let R become R_{new} (with C replaced by C_{new}) and return to 1.

This represents a *class* of algorithms, where different members of the class will be distinguished in terms of the particular ways in which `solve`, `decide1`, and `choose1` are implemented. An algorithm in this class spends a certain amount of time and effort trying to solve R, and based on the results of that effort, either carries on trying to solve R for a little longer, or *changes* R (on the basis that R seems to be, or perhaps has been proven to be, insoluble) and recursively tries to solve the new problem. The function `choose1` must attempt to make a reasonable choice of constraint to change, and make a reasonable change to that constraint, so as to make the new problem solvable in the light of the particular difficulties that seem to have resulted from trying to solve the old one (this is what `decide` looks into).

If our problem spaces are such that we can allow T to be enough for us to always find a solution if there is one (as is the case with CYCLOPS [Navinchandra, 91]), then this algorithm will be effective. If, however, this is not so – which will be typically true in real world design problems – the performance of such an algorithm is likely to be very sensitive to T, and very sensitive to the criteria used in deciding how to alter constraints in step 7. In particular, it seems likely that for most problems, step 2. should be relatively brief, to allow `decide1` to to 'monitor' the state of the problem, continually looking for signs which would suggest that a requirement revision is necessary.

Understanding more about this class of algorithm will come about through more theoretical work and experimentation. In particular, we can consider ISPs which are *inconsistent*, but otherwise complete and precise. Put more accurately, this means investigating instantiations of this general class of requirement revision algorithms which attempt to find *any* solution which fits the requirements in step 1. (Hence, the first solution found will suffice, and so imprecision will not be an issue), and will not attempt to speed-up its performace in step 1 by trying to learn more heuristics. This kind of analysis of the $\mathcal{I}1$ class of algorithms will show how different amounts of K, and different kinds of `decide1` and `choose` functions affect performance of the algorithm.

4.3 An algorithm for incomplete problems

As before, we have a degree of background knowledge K, a problem, in the form of a set of requirements R, and resource parameters T.

Algorithm $\mathcal{I}2$:

1. Input: A problem R, resource parameters T, and background knowledge K.

2. Apply resources T in trying to solve problem R, yielding result S. Eg: apply `solve`(R,T,K), yielding S.

3. If S indicates that a solution has been found, then halt.

4. If not, then execute `decide2`(R,S,T,K), yielding result Q in order to determine in what ways the knowledge accessed by `solve` can be augmented so as to solve the problem.

5. If Q indicates that it is likely that the problem could be satisfactorily solved by expending more resources with the current state of knowledge K, then return to 1. (continuing from where 1. was interrupted).

6. If Q indicates that it is likely that the problem could be solved by finding new operators for `solve` to use, then execute `find-new-operators`(K) and return to 1.

7. If Q indicates that it is likely that the problem could be solved more easily by finding better heuristics for the existing operators of `solve`, then execute `find-new-heuristics`(K), and return to 1.

8. If Q indicates that it is likely that the problem can only be solved by extending K in some other way (finding new operators which `find-new-operators`(K) couldn't, finding new heuristics which `find-new-heuristics`(K) couldn't, or generally making quite radical imporvements and additions to K), then execute `complete`(K) and return to 1.

An algorithm in this class (identified in terms of particular ways of implementing `solve`, `find-new-heuristics`, `find-new-operators`, and `complete`), in a very similar way to $\mathcal{I}1$, first spends a certain amount of effort in having a go at the problem. Then, if the problem is still unsolved, `decide2` looks at the situation, and decides how to find more knowledge in order to help solve the problem. It is interesting to note that Soar deals with incompleteness (and in fact, ill-structuredness and imprecision) largely by a process `complete`(K) (in step 7) which involves asking the user for information (which the user may take several days or weeks to provide) ([Steier, 92]). Our work will involve trying to establish the behaviour of this particular class of algorithm in terms of different amounts and kinds of domain knowledge K, resources T, and problem R.

4.4 An algorithm for imprecise problems

Again, we have a degree of background knowledge K, a problem, in the form of a set of requirements R, and resource bounds T. In order to solve an imprecise problem, K must also include knowledge concerning possible other constraints which can be imposed in the domain, as well as some interpretation of their relative importance (ie: some kind of prioritisation over possible constraints).

Algorithm $\mathcal{I}3$:

1. Input: A problem R, resource bounds T, and background knowledge K.

2. Apply resources T in order to try to solve problem R, yielding result S. Eg: apply `solve`(R,T,K), yielding S.

3. If S is a solution to R, in the sense that there are N possible solutions, none of which differ in any important respect, then halt.

4. If S is not a solution, in that there are N possible solutions to R, which all satisfy R but differ from each other in ways which are important in the domain, then execute `choose2`(K,R,S), yielding result C. Ie: given domain knowledge relevant to the problem, choose a new constraint from that domain which is different from those in R, and which will help to discriminate among the possible solutions S.

5. Let R become $R \cup C$, and return to step 1, continuing from where step 1 was interrupted.

Just as with $\mathcal{I}1$ and $\mathcal{I}2$, this class of algorithm spends some amount of effort in trying to solve a problem, and comes to a point at which it must decide how to continue. Since the only difficulty $\mathcal{I}3$ expects is that there are too many possible solutions which differ in important ways, it either halts if this is not the case, or chooses a discriminating constraint if it is. As with $\mathcal{I}1$ and $\mathcal{I}2$, We plan to look at $\mathcal{I}3$ in my research in more detail, so as to try and characterise ite behaviour with regard to different kinds of input, and different kinds of `choose2` function.

4.5 A general ill-structured problem solver

Our research will continue by looking further into the nature of these classes of algorithms, in terms of both collating examples of similar processes in the literature which seem to be examples of one or more of these, and theoretically investigating the nature and performance of them in (necessarily) simple realisations and domains. In tandem with this line of study, we are looking at computational mechanisms which will underpin a general *computational exploration* process which can handle a wide subset of ISPs. The junctures at which these algorithmic recipes will be 'combined' in a general ISP-solver will be at the level of the underlying decision subprocesses in each algorithm, *ie.*, `solve`, `decide1`, `decide2`, `choose1`, and `choose2`. Work on each of the $\mathcal{I}$-algorithms should yield useful and interesting results concerned with what degrees and types of inconsistency, imprecision, and incompleteness can be handled, and with what degree of efficiency and success, in terms of what kinds of and amounts of background knowledge K are present. Such results will also be qualified in terms of the nature of the space of problem requirements, and the nature of the functions `solve`, `decide1`, `decide2`, and `choose`. However, combining these algorithms to produce a more general process for solving ISPs will probably best not involve simply joining these individual algorithms, for the same reason that it does not make sense to eat a balanced diet by having nothing but fruit one week, and nothing but pulses the next. Rather, the structure of an effective algorithm for basic computational exploration will in large part emerge from considering the aforementioned basic subprocesses, *ie.*, combining them at the level of `solve`, `choose`, and `decide`. (*ie.*, eating a balanced diet by devising sufficiently tasty and varying combinations of the basic food groups to make particular meals).

We can now outline five major kinds of activity, involving these subprocesses, which our previous research (on collaborative design support systems, see [Smithers *et al*, 90], [Logan *et al*, 91], [Logan *et al*, 92]) and current ideas (*eg*: see [Smithers, 92], and [Corne *et al*, 92]) have led us to think are close to the minimal, sufficient set of processes required in ISP-solving:

Solution Construction: The process `solve` can be seen as the niche in our investigation for the generally understood process of solving well-structured problems. The basic job of `solve` is to construct solutions which will hopefully meet a set of requirements R, and its structure will probably follow on from a basic design decision I intend to make (so as to allow knowledge representation to be both flexible and simple): `solve` will, at heart, use production rules (based on a *design grammar* for the particular domain area) to infer solution construction actions, (adding something to the current partial solution, or changing the current partial solution in some way) from details of the current partial solution and the current goals. The rules at the heart of `solve` will be accessed, augmented and changed in various ways in the course of exploration via, especially, the kinds of process used in $\mathcal{I}2$.

Problem Construction : The second major component of `solve` is a *problem construction process*. This is the process which derives a plan of attack for either constructing a solution to R (*ie.*, a plan might involve breaking R down into subproblems and later merging them together) or a more complex plan which includes constructing *k-subproblems* to solve, which serve to find out more knowledge about the problem in terms of heuristics or new solution construction rules. For example, if R concerns designing a kitchen layout, then `solve` may either first construct a plan which simply specifies a series of subproblems to solve and merge, or, if there is likely to be insufficient knowledge present to do this satisfactorily, a *k-subproblem* may be constructed such as "find out what groups of applicances and units can safely be put along wall W". K-subproblems are one means by which exploration can be guided to deal with incompleteness.

Requirement Revision: Central to the three $\mathcal{I}$-algorithms is the process of *requirements revision.* This happens whenever a set of requirements is changed to (attempt to) remove inconsistency, or augmented to enable better discrimination between solutions, or some combination of these two. Dealing with incompleteness also executes a kind of problem revision, but such that the problem is different *with respect to* the knowledge available to the process; *ie.*, the problem becomes easier to solve, rather than changes its form. Nevertheless, my research might extend to areas in which extra knowledge gleaned from dealing with incompleteness is used to revise requirements descriptions in the latter two ways, rather than just failitating the activity of `solve`. The general problem revision process must be carried out via a combined realisation of `decide1`, `decide2`, `choose1`, and `choose2`. Ie: the problem revision process must have access to domain knowledge, as well as the course (history) of the exploration process so far. Requirement revision will most likely be implemented in the form of a hierarchy of requirement revision decision rules (amenable to change and augmentation) which relate the state of play so far with respect to the current set of requirements to requirement revision actions. These actions will form the basis for the *Problem Revision Path* constructed throughout the process, which in turn encapsulates the *validity* of the resulting design solution.

Experiential & Exploratory Learning: Another major process which lies at the core of the $\mathcal{I}$-algorithms is that by which decisions made at any level (solution construction, requirement revision, problem construction) are monitored for their effects, and thus learned from, so as to modify these rules and/or create new ones which improve the quality of the process. This kind of learning is fundamental to designing; even if it is not necessary to deal with incompleteness in a particular problem, it remains necessary to better inform the choice and decision functions for inconsistency and imprecision. We can see this as necessary by imagining, how $\mathcal{I}1$ and $\mathcal{I}2$ might each perform if no learning was involved; it would seem evident that with relatively large problems the amount of requirement revisions (and so the amount of nested recursions of the algorithms) would become excessively large and excessively time-consuming if the process did not at least try to incorporate some method for incrementally improving the efficacy of the choice and decision functions. Each set of rules, then, must be accessible and usable by a general learning mechanism, (perhaps based on classifier-style learning; *ie.*, an algorithm which incrementally modifies rule strengths according to performance, coupled with an algorithm which creates new rules from relatively good old ones and tries them out).

Exploration Management : Somehow, the entire process must be monitored and managed by a central co-ordinator of activites. This co-ordinator, the *exploration management* process, will be responsible for tuning the degrees to which different amounts of resources and effort are applied to different parts of the process. *Eg.*, a simple exploration management strategy, based on the "glueing them together" route, would be simply to exhaust effort on $\mathcal{I}1$, and if that doesn't yield a solution, exhaust effort on $\mathcal{I}2$, and so on. Rather more likely, exploration management will be designed so as to apportion activity according to the nature of the initial problem, and change this apportioning at different stages.

This section, then, has attempted to set out the basic ideas underlying our current research programme. We have noted that three basic recipes for the three basic kinds of ill-structuredness are each worth further study. This study will involve theoretical groundwork and physical implementation of examples of the algorithms, along with survey and collation work which attempts to find examples of such algorithms already implemented in the literature. This study should contribute to the development of ideas and expectations for a more general ISP-solving process. We have also indicated that, orthogonal to the $\mathcal{I}$-algorithms, are five more basic processes which will be important to such an ISP-solving process, and form the junctures at which we should be able to weld together some variants of the three $\mathcal{I}$-algorithms. Our research will therefore involve theoretical and experimental analysis of the three $\mathcal{I}$-algorithms, in conjunction with development of ideas for the orthogonal exploration processes. We then expect to be in a position to implement and experiment with a general *computational exploration* system which effectively handles a non-trivial class of general ill-structured problems. The main experimental problem domain will be augmented 2D layout design (a very common, important and rich domain in AI-in-Design) for office layouts, site layouts, and similar; subsidiary domains for comparative experiements and general investigation will be algorithm design, board game design, and crossword puzzle design.

5 Summary

Our AI/Design research is aimed at gaining enhanced theoretical understanding of ill-structured problems and enhanced techniques for finding effective solutions to them. This involves survey work to tease out contributions to our perspective from a disaparate literature, theoretical work in usefully taxonomising and operationalising the notion of different kinds of ill-structuredness, and practical work aimed at the development of minimal computational techniques which have maximal effectiveness in dealing with ill-structuredness. Our theoretical beginnings involve the notion of *problem revision paths* to capture what it means to effectively solve an ISP, the notion of three major kinds of ill-structuredness, the characterisation of some simple algorithmic recipes for dealing with these kinds of ill-structuredness, and the initial steps in the design of a general ISP-solver. In the longer term, we believe that such ISP-solving systems, especially through their use in design problem solving, will be an excitingly novel kind of system. Since, depending on the particular kind of requirement revision rules (which establish what can be classed as an effective solution, via their sanction of what are deemed allowable requirement revision arcs) we may usually not expect them to produce answers too closely related to the problems we input (typical of designers in general), we will be able to deploy such systems at much earlier stages in the real-world problem-solving process than current systems, and may be surprised and delighted by their individual, idiosyncratic, and non-deterministic solutions.

References

[Amarel, 90] Amarel, S. *Problem Solving*, Tehnical Report LSCR-TR-152, Laboratory for Computer Science Research, Busch Campus, Rutgers University, New Brunswick, NJ, 1990. (also a chapter in the 2nd Edition of *The Encyclopedia of AI*, Shapiro (ed.))

[Boden, 90] Boden, M.A. *The Creative Mind: Myths and Mechanisms*, George Weidenfeld & Nicolson Ltd, London, pages 134–181, 1990.

[Colgan & Spence, 91] Colgan, L. and Spence, R. *Cognitive Modelling of Electronic Design*, in *Artificial Intelligence in Design '91*, J. Gero (ed.), Butterworth-Heinemann, pages 543–559, 1991.

[Corne *et al*, 92] Corne, D., Logan, B. & Smithers, T. *A Fundamental Tradeoff in Intelligent Design Support*, in *Proceedings of the 6th International Conference on Systems Research, Informatics, and*

Cybernetics, Volume 1: Advances in Computer-Based Design Environments, J. Pohl (ed.), ISBN 0–921836–00-7, 1992.

[Davies & Simplicio-Filho, 92] Davies, S.P. and Simplicio-Filho, F. *Opportunistic and Goal-Oriented Behaviour in Software Design*, in *Artificial Intelligence in Design 92* J. Gero (ed.), Kluwer Academic Publishers, pages 839–860, 1992.

[Goel & Pirolli, 92] Goel, V., and Pirolli, P. *The Structure of Design Problem Spaces*, in *Cognitive Science*, vol 16 no. 3, 1992.

[Logan *et al*, 91] *Being Economical with the Truth: assumption-based context management in the Edinburgh Designer System*, in *Artificial Intelligence in Design 91*, J. Gero (ed.), Butterworth-Heinemann, pages 423–446, 1991.

[Logan *et al*, 92] Logan, B., Corne, D. and Smithers, T. *Enduring support: on defeasible reasoning in design support systems*, in *Artificial Intelligence in Design '92*, J. Gero (ed.), Kluwer Academic Publishers, pages 433–454, 1992.

[Maher & Li, 92] Maher, M.L. and Li, H. *Automatically Learning Preliminary Design Knowledge from Design Examples*, Microcomputers in Civil Engineering vol 7 no. 1, pages 73–80, 1992.

[Navinchandra, 91] Navinchandra, D. *Exploration and Innovation in Design: Towards a Computational Model*, Springer Verlag, New York, 1991.

[Qian & Gero, 92] Qian, L. and Gero, J.S. *A Design Support System Using Analogy*, in *Artificial Intelligence in Design '92*, J. Gero (ed.), Kluwer Academic Publishers, pages 795–813, 1992.

[Simon, 69] Simon, H. A., *The Sciences of The Artificial*, MIT Press, 1969.

[Simon, 73] Simon, H. A., *The Structure of Ill-Structured Problems*, Artificial Intelligence, 4, pp 181–201, 1973.

[Smithers *et al*, 90] Smithers, T., Conkie,A., Doheny, J., Logan, B., Millington, K., and Tang, M.X. *Design as Intelligent Behaviour: An AI in Design Research Programme*, in *Artificial Intelligence in Engineering* vol 5, number 2, pages 78–109, 1990.

[Smithers & Troxell, 90] Smithers, T. and Troxell, W. *Design is Intelligent Behaviour, but What's the Formalism?*, in Artificial Intelligence in Engineering, Design, Analysis and Manufacturing, vol 2 no. 4, pp 89–98, 1990.

[Smithers, 92] Smithers, T. *Design as Exploration: Puzzle Making and Puzzle Solving*, in D. Corne (ed.) *Workshop on Search-Based and Exploration-Based Models of the Design Process*, EDAI DRG Working Paper D4, pages 1—21, 1992.

[Sriram *et al*, 89] Sriram, D., Stephanopoulos, G., Logcher, R., Gossard, D., Groleau, N., Serrano, D., and Navinchandra, D. *Knowledge-Based System Applications in Engineering Design: Research at MIT*, AI Magazine, pages 79–96, Fall 1989.

[Steier, 92] Steier, D. *Design as Multiple Problem Space Search*, in D. Corne (ed.) *Workshop on Search-Based and Exploration-Based Models of the Design Process*, EDAI DRG Working Paper D4, pages 28—29, 1992.

[Steinberg & Ling, 90] Steinberg, L., and Ling, R. *A Priori Knowledge of Structure vs Constraint Propogation: One Fragment of a Science of Design*, AI/Design Project Working Paper No. 164, Dept. of Computer Science, Rutgers University, New Brunswick, NJ, 1990.

[Tham & Gero, 92] Tham, K.W. and Gero, J.S. *PROBER — A Design System Based on Design Prototypes*, in *Artificial Intelligence in Design '92*, J. Gero (ed.), Kluwer Academic Publishers, pages 657–675, 1992.

[Tong, 90] Tong, C. *Knowledge-Based Design as an Engineering Science*, Rutgers University research report, August 1990, WP-TR-167.

[Visser, 91] Visser, W. *The Cognitive Psychology Viewpoint on Design: Examples From Empirical Studies*, in *Artificial Intelligence in Design 92* J. Gero (ed.), Kluwer Academic Publishers, pages 505–524, 1991.

[Wanken, 91] Wanken, A. *The Design of Abstract Graphics*, MSc Dissertation, University of Edinburgh Department of Artificial Intelligence, 1991.

[Westerberg *et al*, 89] Westerberg, A., Grossmann, I., Talukdar, S., Prinz, F., Fenves, S. and Maher, M.L. *Applications of AI in Design Research at Carnegie Mellon University's EDRC*, in *Artificial Intelligence in Design: proceedings of the 4th International Conference on the Applications of Artificial Intelligence in Engineering, Cambridge, UK, 1989*, J. Gero (ed.), Springer-Verlag, pages 335—362, 1989.

[Whitefield & Warren, 89] Whitefield, A. & Warren, C. *A Blackboard Framework for Modelling Designers' Behaviour*, Design Studies Vol 10 No. 3, July 1989, Butterworth & Co.