ENDURING SUPPORT

On Defeasible Reasoning in Design Support Systems

BRIAN LOGAN and DAVID CORNE
Department of Artificial Intelligence
University of Edinburgh
Edinburgh, UK

and

TIM SMITHERS[*]
Artificial Intelligence Laboratory
Vrije Universiteit Brussel
Brussels, Belgium

Abstract. The task of a design support system is conventionally conceived as one of providing the designer with solutions to specific parts of a design problem. In this paper we argue that this approach is fundamentally flawed. We identify two main modalities of support: the production of *necessary* consequences and the production of *possible* consequences of the current design description, and discuss the problem of devising an architecture capable of providing such support using the Edinburgh Designer System (EDS) as an example. We describe the difficulties inherent in integrating the derivation of possible consequences into the architecture of EDS and argue that while in principle such difficulties can be overcome, in practice the goal of providing globally consistent solutions to particular parts of the design problem is unattainable. We propose a different approach in which the design support system *explores* the consequences of various design decisions. The results of this exploration are represented as counterfactual conditionals which we believe more closely approximates the information required by a designer.

Key words: exploration – defeasible reasoning – design support systems

1. Introduction

We take designing things to be a particular kind of intelligent behaviour. We further presume that the nature of the process underlying this kind of behaviour has a general form common to all instances and types of designing. For us, then, what human designers do when they design things is to be taken as a way of implementing this process—possibly the only way at the moment—but not as a way of defining it. In taking this stance we are able to identify the design process in the abstract, independent of its human implementations, and thus to ask how it might be understood in computational terms.

The long term aim of our AI in Design research is a computational theory of design process which can be used to both explain design behaviour and to predict design performance in all its forms; natural or artificial. In pursuit of this aim we have developed what we call the exploration-based model of design process (Smithers & Troxell, 1990), and have been attempting to verify and develop it by

[*] SWIFT Professor of AI on leave from the Department of Artificial Intelligence, University of Edinburgh.

building experimental design support systems (Buck et al., 1991; Smithers et al., 1990).

Our research method involves investigating the design process by trying to build design support systems that actively participate in design tasks with human designers. In this way we are able to control how much and which parts of the overall design process we try to directly investigate at any time. It also emphasises the problems of human-computer interaction which we inevitably have to face up to. This approach contrasts with other methodologies found in AI in Design research which attempt to build automated design systems. The role of of the design support systems that we build and experiment with can be identified within the following framework used to organise our research programme:

1. *Exploration-based model of design process*, in which we are seeking to develop a general model of the design process which attempts to identify and relate all its essential characteristics and properties.
2. *Domain models of design process*, in which we are seeking to understand how the general model is to be specialized to model the design process in a particular domain. This typically involves identifying the nature, type, and amounts of knowledge required to pursue design tasks in the domain, and identifying the nature, type, and scope of the activities engaged in when solving design tasks in the domain. In this way, what are called routine, innovative, and original design tasks can be identified as different configurations of the general model.
3. *Design support models*, in which we are trying to identify those of the activities identified by the domain models that can be supported in an integrated and coherent way using a computer-based support system, and to identify that part of the knowledge required to design in the domain which could be represented within such a support system as part of its support activities.
4. *Design support system architectures*, in which we are attempting to identify the functional components, organisational structure, and control capabilities required of a computer-based design support system that can meet the needs of human designers operating in a particular domain.
5. *Design support system implementation techniques*, in which we are seeking to identify and develop computational techniques and implementation approaches suitable for the building of effective experimental design support systems.

In this paper we discuss the problem of devising and implementing an architecture for a design support system capable of meeting the needs of human designers operating in a particular domain. We focus on a particular kind of support commonly advocated in the literature—the derivation of *possible consequences* from a partial design description, i.e., providing advice or assistance in solving parts of a design problem. We describe the difficulties inherent in integrating the derivation of possible consequences into the architecture of a particular design support system, the Edinburgh Designer System, and argue that while in principle such difficulties can be overcome, in practice the goal of providing globally consistent solutions to particular parts of the design problem is unattainable. More generally, our approach to this problem will hopefully serve to illustrate our research methodology and the kinds of understanding it produces. To set the context and to motivate the particular problem we will be concerned with, we first briefly reiterate some essential aspects of the exploration-based model of design and two characteristics of design support models that we take to be domain independent to a large extent.

2. The Exploration-based Model of Design

The exploration-based model of design process sets out to address the inherently ill-structured nature of design tasks head on. It characterises the design process as one that starts with an initial requirement description that is incomplete and possibly inconsistent. In many cases the stated objectives are in direct conflict with one another and the designer cannot simply optimise one requirement without suffering losses elsewhere. Different trade-offs between the requirements result in a whole range of acceptable solutions, each likely to prove more or less satisfactory under different interpretations of the requirements. It is the very inter-relatedness of all these factors which is the essence of design problems rather than the isolated factors themselves, and it is the identification and structuring of relationships between these criteria that forms the basis for the design process (Lawson, 1980). The design process cannot therefore be adequately characterised as a *search* process in which the task is essentially one of selection or optimisation over a completely defined space of alternatives.

The initial requirement description is analysed into a final complete and consistent requirement description via the synthesis of possible design descriptions that satisfy some or all of the identified requirements. These descriptions are typically not serially ordered and are intended to span the space of possible designs thought relevant to the task in hand. This process of analysis-through-synthesis necessarily involves the discovery of the structure of the problem to be solved. As possible solutions are constructed and developed they provide an increasingly detailed context against which to test the designer's hypotheses, and the evaluation of a proposal can result in the discovery of previously unrecognised relationships and criteria. Solutions to particular sub-problems are apt to be disturbed or undone at a later stage when new aspects are attended to and the considerations leading to the original solution are forgotten or not noticed. As a result, while the final solution may satisfy all the requirements that are evoked when it is tested, it may violate some of the requirements that were imposed (and temporarily satisfied) at an earlier stage in the design. These explorations help the designer appreciate which requirements may be most readily achieved and those that may be neglected without loss. As part of this process, the designer learns which criterion values will achieve the design goals and how much variation of these values can be tolerated while still achieving acceptable performance. The designer also discovers the implications of achieving the current goal, and any other decisions required to make the attainment of these goals consistent with the existing solution. A large part of the design process is thus devoted to discovering the nature and scope of the problem set by the requirement description which itself changes as exploration takes place. See (Smithers & Troxell, 1990) for more details.

3. A Model of Design Support

The design process outlined above involves a continual cycle of generating alternative design solutions or part solutions and the derivation of the consequences of these solutions in an attempt to understand their implications for other design criteria. These alternatives may embody radically different approaches to the problem or they may be variations on a common theme or both. Each of these solutions may in turn be broken down into a set of simpler sub-problems, (for example, the design of an assembly may be reduced to the design of its constituent components) together with the problem of integrating the resulting part solutions, or it may be broken down into a set of associated functions. Of particular concern are inconsistencies which arise in attempting to satisfy multiple goals, for example when the derived performance fails to meet the design requirements or when the proposed solution implies two or more values for a given design parameter.

We can therefore identify two important aspects of the support required when carrying out design tasks:

1. *Strategic support* for the overall control of the design process. A major task of any design support system is one of complexity management: controlling the generation of alternative solutions to particular sub-problems; comparing the alternatives; selecting the most promising candidates for further development; and revising and refining the requirement description.
2. *Tactical support* for the construction and development of alternative design solutions. The system should provide support for context specific inference based on design decisions, constraints and the requirements defining the design problem, together with physical laws, heuristics and other domain knowledge relating the parameters of the design. In addition the system should, where possible, provide assistance in solving particular design problems, drawing on the large amounts of knowledge encoded in design handbooks, codes of practice and in the expertise of individual designers.

In an previous paper (Logan, Millington & Smithers, 1991) we discussed some of the problems of providing strategic support. In this paper we concentrate on the problems of providing tactical support within the blackboard-based system architecture that we have been developing for the past seven years, see (Buck et al., 1991; Smithers, 1987; Smithers et al., 1990). This system is called the Edinburgh Designer System or EDS for short.

4. System Architecture

EDS consists of four principal subsystems which together embody the model of design support outlined above: *knowledge representation*; *inference*; *consistency maintenance*; and *context management*. In this section we briefly describe the architecture of EDS and how it supports the exploration of the space of possible designs. In the following sections we concentrate primarily on the consistency maintenance and context management sub-systems of EDS and their role in providing tactical support. Context management, knowledge representation and the overall design of the system are discussed in more detail elsewhere (Logan, Millington & Smithers, 1991; Logan & Smithers, 1992; Smithers et al., 1990).

EDS represents domain knowledge in a structured knowledge-based called the Domain Knowledge Base (DKB). This contains definitions of domain objects, called *module classes*, related by *part_of* and *kind_of* relations. Each module class declares a set of parameters, variables and constraints which define a particular class or type of object. This knowledge of the domain is used by a series of inference engines or support systems to infer the consequences of the designer's decisions. There are two main kinds of support system: *general purpose support systems* (GPSSs) and *special purpose support systems* (SPSSs). General purpose support systems use constraint propagation techniques to infer the *necessary* consequences of the designer's decisions. In general these sub-systems are invoked automatically and can be allowed to proceed relatively unhampered by the designer with the sole constraint that they should refrain from re-deriving information already known about the design. Special purpose support systems, on the other hand, advise on how to solve particular design problems. In general these sub-systems offer *possible* consequences, often in the form of one or more design decisions which extend the current design description. They tend to be goal directed and are usually invoked explicitly by the user who then interacts with the support system to reduce the number of options until a single solution is found.

Control of the interactions between the support systems is in the style of a blackboard system (Hayes-Roth, 1985). In the blackboard model the knowledge required to perform a particular task is partitioned into a series of knowledge sources (KSs) each of which performs a particular sub-task. The current state of the task and the results produced by the knowledge sources are recorded in a global database or blackboard. The knowledge sources use information on the blackboard to derive new information using algorithmic procedures or heuristic rules. The system maintains an agenda of knowledge source activation records (KSARs) which identify possible inferences the system could perform. The decision as to which knowledge source to apply in any particular situation is determined dynamically based on the current solution state (and in particular the latest additions to the blackboard) and on the ability of the knowledge sources to contribute to the developing solution. In EDS each support system is implemented as one or more knowledge sources which derive consequences of the current design description represented on the blackboard.

The system pays particular attention to any inconsistencies derived by the knowledge sources as these are often indicative of inconsistencies in the problem requirements or problems with the proposed solution. A design description must be consistent if it is to refer to anything. At the same time we have to recognise that inconsistencies are inevitable—a design description is typically inconsistent for much of its history as the designer explores the space of possible designs attempting to meet the various design requirements. However all a contradiction between two design decisions indicates is that any inference which depends on both decisions is of no value; it is still important to draw inferences from each of the decisions independently.

The desire to derive as much information as possible from inconsistent design descriptions led to the adoption of an assumption-based truth maintenance system (ATMS) for EDS (de Kleer, 1986a). The ATMS builds and maintains a dynamic datastructure, the Design Description Document (DDD), and provides an interface between the contents of the DDD and the other sub-systems, passing out relevant pieces of information to them as required and incorporating new information which it receives from them into the dependency structure. Each piece of information on the blackboard is associated with with an ATMS *node*. A datum p is said to have a *justification* if there exists a set $q_1, \ldots, q_n$ of nodes from which it can be derived. The set $q_1, \ldots, q_n$ are termed the *antecedents* of the justification and p is termed the consequent. A subset of the data, called *assumptions*, are taken as primitive and everything else is derived from them. In EDS these represent the basic design decisions made by the user. By tracing backwards through the supporting justifications, the ATMS identifies the set(s) of assumptions (i.e. design decisions) on which a datum ultimately depends. Such a set of assumptions is called an *environment*. The set of environments in which a datum is known to be derivable is is called its *label*. A datum which has a non-empty label is said to have support, i.e., it can be consistently derived from the assumptions forming each of the environment(s) in its label. If the designer's decisions subsequently turn out to be mutually inconsistent, the ATMS restores consistency by deleting from the label of each datum node all of the environments which contain the inconsistent assumptions. Data which are only derivable in such an environment (i.e., they can only be derived from an inconsistent set of assumptions) become unsupported and hence cannot form the basis of further inferences.

The central role of the ATMS in the system architecture has resulted in a number of extensions to the conventional ATMS model. For example, the ATMS truth maintains the knowledge source activation records within the DDD on an equal footing with existing data and its justifications. This automatically removes KSARs whose antecedents are discovered to be inconsistent. Similarly, blackboard systems are usually designed and implemented as 'single context' problem-solving systems in which the knowledge sources work together to construct one consistent solution. Truth maintaining the blackboard has therefore resulted in a number of departures from the conventional practice in blackboard systems, notably the absence of deletions (except for KSARs) or amendments as it is unclear how these fit into an assumption-based truth maintenance scheme.

Design proceeds by creating instances of module classes and assigning values to their parameters to define one or more possible solutions.[1] When the user makes an assumption one (or more) datum nodes are created to hold the new information. This information is examined by the knowledge sources to see if it, together with any information already assumed or derived, can be used to make further inferences. If a knowledge source is able to make an inference, it generates a bid in the form of a knowledge source activation record which is scored and merged into the agenda. When the KSAR reaches the front of the agenda, it is executed and the results are claimed into the ATMS. This information may in turn form the basis for a new round of bids and this cycle continues until no executable KSARs remain in the agenda. As new values for parameters or bounds on them are assumed or derived, consistency checks are performed between constraints and values by the *valueConflict* KS. Conflicts result in the creation of a justification for the distinguished node $\langle false \rangle$ recording the fact that the assumptions involved are mutually inconsistent and causes the ATMS to partition the assumptions into mutually consistent sets. If there is no conflict, EDS marks this by justifying the datum $\langle consistent \rangle$ and proceeding as usual. The user is viewed as a knowledge source whose 'bids' are always processed first. This allows the system to follow several lines of reasoning as it attempts to infer the consequences of the user's design decisions, while still giving priority to user input.

5. General Purpose Support Systems

A general purpose support system (GPSS) is a reasoning system which performs inferences common to many of the sub-tasks which together constitute the overall design process. Note that 'general' in this sense does not mean that such inferences are necessarily useful in other design domains, although they may be. There are currently four support systems (or 'engines') implementing general purpose support capabilities within EDS (Smithers et al., 1990):

1. the *Evaluation Engine* which handles value propagation, constraint satisfaction and expression simplification;
2. the *Algebraic Manipulation Engine* (AME) which takes an arbitrary set of equations and solves them for any number of variables in which they are linear;
3. the *Relation Manipulation Engine* (RME) which performs value interpolation and relational operations on tabular data; and
4. the *Spatial Reasoning Engine* (SRE) which performs spatial inferencing.

[1] This is an oversimplification—the user can also define new parameters and constraints and assemble novel designs from existing modules.

These are implemented as 35 knowledge sources.[2] There are also two knowledge sources which do not form part of any support system: the *Direct Inference System*, a simple rule interpreter, which can be used to implement user-defined knowledge sources and support systems; and the *USER-KS* which handles user input.

The general purpose support systems can be subdivided into two main types: those which are *user-invoked* and those which are *system-invoked*, i.e., they are triggered automatically by information being claimed into the DDD. This distinction is based largely on pragmatic considerations, and in particular the computational costs associated with the support system. Ideally, all the support systems would be invoked automatically. Our objective is to assist the designer in exploring the space of possible designs by providing information useful in determining the future development of the current design solution, rather than (as conventionally happens) by providing the means to obtain such information. However our understanding of the design process at the level of an individual designer solving a particular design problem is insufficient to determine which of the many inferences the system could make are most appropriate at any given point in the problem solving process. This basic difficulty is compounded by two additional problems:

1. the design description is constantly changing, both as a consequence of assumptions made by the designer and information derived by the system from the current design description; and
2. any inferences made by the system may subsequently be invalidated if the underlying assumptions are discovered to be inconsistent.

One strategy would be to try to derive everything we can about the design. While this may result in the derivation of some useless information, it allows us to have confidence in the information we do produce, as any inconsistencies implicit in the design description which the system is capable of finding will be discovered. This is the rationale underlying the choice of the blackboard and its opportunistic control strategy. At each cycle, the system applies the knowledge sources to the current design description, reviewing any as yet uncompleted work in the light of what was discovered at the last cycle—inconsistencies, 'more interesting' facts etc.—and adjusts its inference priorities accordingly. However this approach is not practical in its pure form. While the KSs are selected with a view to producing useful consequences, not all of them will be equally useful in a given situation, and at any point there will be many more inferences we can make than we have the resources to make. We therefore compromise. If the computational costs associated with a particular support system are high, the decision to invoke it is typically left to the user due to the difficulty of determining *a priori* the relevance of the information produced to the user's current interests and objectives. On the other hand, if the computational costs are low, the support system is typically invoked automatically,

[2] The definition of a knowledge source in the blackboard literature is rather vague. In another interpretation, we could say there are 4 knowledge sources, one for each support system, each of which consists of a number of rules or 'methods'. However such an interpretation obscures the relationship between the KSs and the justifications they produce.

even if the results may not be of immediate interest to the user. Hopefully this will also reveal any inconsistencies before the computationally expensive support systems are invoked by the user. The difficulties associated with this heuristic are part of a larger problem involving the determination of the context of design tasks and the control of inference which is considered in (Logan, Millington & Smithers, 1991). In EDS, the Evaluation Engine and the user-defined support systems built using the Direct Inference System are automatically invoked by the system, while the AME, RME and SRE are all user-invoked.

6. Special Purpose Support Systems

Unlike a general purpose support system, a special purpose support system or *specialist* is a reasoning system which provides support for a particular design task, such as design for manufacture, design for maintenance or cost analysis. This may involve advising on particular aspects of design solutions with respect to, for example, their cost or manufacturing implications, or suggesting general strategies for resolving identified and well-understood conflicts. In general, this requires specialist knowledge which is unique to both the task and domain. Such support systems tend to be goal directed and would typically operate under the close supervision of the user.

In EDS special purpose support systems are implemented using a Prolog meta-language or *shell*. The EDS shell is a backward-chaining rule-based inference system which provides a framework for building specialised design support systems based on design heuristics in the form of production rules. This approach is natural given the architecture of EDS and is in accordance with much of the existing work on 'support systems' for design. The production rules are constructed using, a library of Prolog clauses which allow examination of the DDD, DKB map, ATMS justification structure and the Prolog versions of any loaded modules. These clauses act like normal Prolog clauses, they succeed or fail and, if appropriate, instantiate variables.

However there are a number of problems in integrating these SPSSs into the EDS architecture.[3] While the EDS shell provides a flexible environment for the definition of new support systems, such systems may violate some of the implicit assumption on which the ATMS-blackboard model is based. Two main problems can be identified: determining what advice to give; and truth maintaining the results. The former involves finding out which design alternatives are consistent with the current solution, while the latter involves ensuring that any inconsistencies which arise in the future are detected.

The task of a special purpose support system is typically to find a solution to

[3] There are currently no special purpose support systems in EDS. A single SPSS which advised on the selection of appropriate subclasses for the transmission module on the basis of the values of the transmission parameters was implemented as a demonstration. However this system does not form part of the current version of EDS and no other special purpose support systems have been developed.

a particular design problem. To be useful such proposals must be selective; it is rarely possible or even desirable to enumerate all possible ways of solving a design problem. Presumably such a solution should be consistent with the current design context, otherwise the task of the specialist is trivial.[4] However the consistency or otherwise of a particular datum with the current state of the DDD cannot be determined by the KS that derived it. In the current implementation, the blackboard is suspended when a knowledge source is running. As a result, the consequences of any claims made by the KS are not derived until the next blackboard cycle. There is no general mechanism to request the proof of a particular goal. If a KS wants a piece of work done by another KS it must know how to claim an appropriate KSAR. Even if it can generate a KSAR, the calling KS cannot wait while the sub-goal is proved. If it cannot proceed without the information, it must fail and rely on being re-invoked by its trigger when the new information becomes available.

Even if it were possible for an SPSS to invoke the blackboard and have any derived results checked for consistency without losing overall control of the system, such an interaction model is incompatible with the 'opportunistic' control strategy of the ATMS-blackboard. Furthermore, if an inconsistency is detected, the system must backtrack, deleting any claims and/or assumptions made prior to the discovery of the inconsistency. At present while assumptions can be deleted claims cannot. Any label propagation performed prior to the detection of the inconsistency must also be undone and the new nogood set(s) deleted from the label of the false node. It is not clear how this could be achieved.

Clearly the advice offered by a specialist may have considerable influence on the design process and should therefore form part of the design history. However even if the specialist can determine what advice to give the system cannot simply extend the design description by making additional assumptions which achieve the goal. Even if the additional assumptions are consistent with the rest of the design description this is typically not acceptable as the system cannot explain *why* the solution is what it is. It simply records the fact that if the design description is extended in a certain way, a particular consequence follows. To *justify* such an assumption, it would be necessary to record why it was preferred to other possible extensions to the design description. This reflects the basic asymmetry between the system and the user; while the system is expected to be able to explain (and hence justify) everything it does, the user is often incapable of explaining why they investigated a particular alternative or enumerating their reasons for preferring one alternative to another. Without knowledge of the assumptions justifying the output of a specialist we cannot detect potential inconsistencies between the user's and the specialist's assumptions. Clearly what can be derived is limited by the system's knowledge of potential inconsistencies (in the form of constraints relating parameter values) and what it has been told by the user. While we cannot trap all

[4] There is also the problem of what the solution is to be consistent with if the current design description is itself inconsistent.

inconsistencies in the user's assumptions, it seems prudent to avoid compounding these problems by allowing the system to make unsupported assumptions.

If the specialist simply offers 'advice' in the form output presented to the user or claimed into the the ATMS (e.g. "use material x") rather than attempting to extend the design description the position is even worse. If the advice is not claimed into the ATMS then it will not form part of the design history and it will be impossible to determine if the advice should ever be withdrawn as a consequence of changes to the current design context. Even if the advice is truth maintained and the specialist is explicit about the supporting assumptions on which the advice is based, if the user subsequently acts on the advice by making the corresponding assumption(s) (i.e., uses material x in the current design description), any inconsistencies which arise later may not be detected because the supporting assumptions justify the the result claimed by the specialist, not the user's assumption(s). Extending the design description therefore seems preferable, even though this will typically result in other KSs being invoked to derive the necessary consequences of the specialist's advice. Indeed this could be seen as a positive advantage since it means that the implications of the new decision are derived automatically.

Typically such decisions depend on the presence or absence of certain design characteristics. If all the assumptions required by the specialist to derive its advice form part of the current design description there is no difficulty. Such inferences can be viewed as necessary consequences of the current design solution and the decision to use the specialist. However if the problem cannot be solved using only the facts currently known about the design, the specialist musk ask the user for the information it requires or make *additional* assumptions about the future development of the design. Alternatively, the specialist can use the *absence* of certain features in the current design description to justify its conclusions. Note that this still involves the system 'completing' the design; it assumes that because P is currently not derivable, it never will be derivable.[5] For example, if an SPSS is capable of deriving one of a set of disjunctive choices based on the absence of information from the DDD, the dependency on this lack of information must be recorded. In a rule-based SPSS (defined using the EDS shell) the ordering of the rules imply additional constraints on any resulting derivation. Given a left-to-right depth-first execution strategy, the rules:

$$\begin{aligned}&A \wedge B \rightarrow D\\&A \wedge C \rightarrow E\end{aligned} \tag{1}$$

are equivalent to:

$$\begin{aligned}&A \wedge B \rightarrow D\\&A \wedge C \wedge \neg Pr(B) \rightarrow E\end{aligned} \tag{2}$$

[5] This is called the 'contingent nature of absence' in (Tsang, 1991). Tsang states: "decision making may not be based on absence of constituents since any absent constituent may be present at a later stage in the process."

where Pr denotes the object-level provability relation. In Equation (2) the dependence of E on the inability to prove B is made explicit. This requires a meta-level statement about the derivability or otherwise of B to express the implicit reliance on negation as failure at the object-level. These constraints form a set of additional antecedents to the justification supporting any derived result.

This dependency cannot be expressed directly using logical negation at the object-level. The formula:

$$A \wedge C \wedge \neg B \rightarrow E \tag{3}$$

states that B must be *false* for E to be derivable, whereas Equation (2) states only that B should not be *derivable*. If we are willing to interpret negation as failure at the meta-level, we can obtain the same effect at the object-level by using the *closed world assumption* (or equivalently by using logical equivalence in definitions) (Clark, 1977).[6] While the *valueConflict* KS effectively enforces a (strict) form of the closed world assumption for instantiated variables allowing the derivation of $\langle \mathit{false} \rangle$ from $colour(block1, blue)$ and $colour(block1, red)$, the DKB syntax lacks the power of full first-order logic. As a result it is impossible to express universally quantified constraints such as $(\forall x)(\neg B(x))$. The problem is not one of detecting an inconsistency between two values, but recording the dependence of the result on the absence of *any* value for $B(x)$.

Such inferences effectively require a default logic; whether D or E is derivable depends on current state the DDD and what can be inferred from it. For example, if B is subsequently assumed or becomes derivable in the current environment, E will become inconsistent and the environment must be partitioned. However, as we have seen, even if we could represent the fact that B cannot be proved, we cannot establish whether it can in fact be proved without abandoning or seriously compromising the blackboard model of control, unless the KS can establish its own antecedents without assistance from the rest of the system.

Soliciting information from the user is also somewhat problematic. If the information supplied by the user leads to the derivation of an inconsistency which undermines the current state of the KS, what should the KS do? Should the inconsistency be recorded—after all, it involves a user assumption—and the KS fail? Alternatively, should the inconsistency be viewed as a 'possible inconsistency'—since a SPSS by definition only infers possible consequences—and the inference discarded? The answer to this question seems to depend on whether information solicited from the user 'counts as' an assumption (would the user have made the same assumption if the specialist had not asked for the information), and more generally whether negative information in the form of an inconsistent partial solution is regarded as useful information which should form part of the design history. If it is, then even if a consistent solution can be found, any inconsistent alternatives

[6] Note that this changes the semantics; $\neg Pr(B)$ becomes equivalent to $\neg B$. If this is to be consistent, Pr must accurately represent the object-level proof relation, i.e., $\neg Pr(B) \leftrightarrow Pr(\neg B)$ (Kowalski, 1979).

tried before the consistent solution was derived should also form part of the design history as a record of what didn't work and a partial justification of why the derived result was produced. Keeping a record of inconsistent alternatives may be the only principled approach, as any attempt to prove the consistency of a particular result is relative to some resource bound and subsequent processing may result in the derivation of an inconsistency.

7. The ATMS and Blackboard Models of Inference

Superficially, many of these difficulties appear to be the result of a conflict between the ATMS and blackboard models of inference. In the ATMS model described by de Kleer (1984), a problem-solver deduces new data from old through the application of rules or 'consumers'. Associated with each consumer is a set of data which form the antecedents of the consumer. A consumer is invoked when all of its antecedents are believed in the current context. Every inference resulting from a consumer invocation is recorded as a justification, which explicitly represents the dependency of the derived result on the antecedents of the consumer. In contrast, the blackboard model is much less specific about what constitutes an inference. In the blackboard model, the knowledge needed to solve the problem is partitioned into a number of independent chunks roughly corresponding to areas of specialisation within the task, and which divide the overall problem into a series of loosely coupled sub-tasks. These tasks or areas of expertise are implemented as 'knowledge sources'; inference systems or procedures capable of solving problems in their particular domain. Given such a broad characterisation of inference, a precise definition of what constitutes a knowledge source is difficult. Nii (1986) defines a a knowledge source as anything which "makes change(s) to blackboard object(s)".

There are obvious differences between the single rule application of the ATMS and the less easily characterised inferences performed by knowledge sources in the blackboard model. Whereas the ATMS model is monotonic with respect to proof, i.e., once a proposition has been derived it may not be deleted or changed, the blackboard model permits arbitrary modifications and deletions of blackboard objects. As a result it has been necessary to place a number of restrictions on the operation of the blackboard to allow truth maintenance. The most important is probably the absence of deletion and amendment. In the ATMS model, problem-solving is always carried out relative to a particular context. However the deletion of an item may have consequences for contexts other than the context in which the decision to delete the item was made. Amending an item suffers from the same problem (in that it implicitly involves deletion), unless the amendment entails the item in all the contexts in which the item is currently believed. Even with these restrictions a number of problems remain, particularly with respect to the integration of special purpose support systems, and further restrictions are necessary if the ATMS-blackboard model is to be coherent.

SPSSs infer possible rather than necessary consequences, which may involve relying on the absence of information in a particular context to draw a conclusion. To characterise this kind of inference requires a default logic. Even if we had some way of establishing the consistency of a particular datum, in the current implementation it is impossible to record the dependency of derived results on the continued absence of a particular set of assumptions. To do so would require a non-monotonic ATMS. A 'non-monotonic' ATMS is a monotonic ATMS augmented with 'out-nodes'; nodes representing propositions for which proofs cannot be found. An out-node *out-p* is defined to hold in the context of an environment E if and only if p is not a member of the context of E. We can view this definition as a general principle by which environments are completed with sets of out-nodes. The extension-base (the set of out-assumptions which hold for a given environment) and hence the context of an environment is non-monotonic with respect to assumptions and justifications. Since *out-p* holds in an environment E only when p is not a member of the context of E, the derivation or assumption of p results in the retraction of *out-p* and everything which depends on it. The ATMS used by EDS is monotonic. While it is possible to represent normal defaults using the approach proposed by de Kleer (1986c) (given a suitable extension to the DKB syntax to handle quantified negation) it computes incorrect labels for non-normal defaults (Junker, 1989).

As a result, unless care is taken, knowledge sources may violate the assumptions on which the ATMS-blackboard is based. If the inferences performed by the system are to be truth maintained, a number of limitations have to be placed on the design and implementation of knowledge sources:[7]

1. all assumptions used in deriving a result must form part of its justification;
2. a knowledge source may not use the absence of information to justify a conclusion; and
3. a knowledge source may not invoke another knowledge source which might render its antecedents inconsistent.

Condition (1) requires that a knowledge source be *functional*, i.e., the consequent of any resulting justification must be a function of its antecedents and nothing but its antecedents. Condition (2) follows immediately from Condition (1) and the fact that the ATMS-blackboard is monotonic. Condition (3) is a consequence of the inability of the ATMS-blackboard to backtrack. Since the ATMS-blackboard is monotonic, the only way to implement defeasible reasoning is to explicitly manipulate the contents of the DDD to reflect what is currently believed. Leaving aside the difficulties of backtracking over label propagation, such an approach violates the basic ATMS model.

8. The Role of Defeasible Reasoning in Design

If the analysis presented above is correct, we seem to be faced with a choice: either we accept the constraints on support systems presented in the previous section or

[7] These limitations are similar to the restrictions on consumers identified by de Kleer (1986b).

we have to extend the current system in a number of ways. Neither of these options is very attractive. If we stay within the bounds of what can be truth-maintained by the ATMS-blackboard the types of inferences that can be performed by special purpose support systems are very limited. On the other hand, implementing SPSSs as originally envisaged would require fundamental changes to the architecture of the system.

However we believe that these difficulties are indicative of more fundamental problems with our knowledge level analysis (the model of design support). We feel that extending the current implementation in the ways outlined above would in fact run counter to our exploration-based model of the design process, in placing total responsibility for a solution on a single sub-system. In effect it amounts to design without exploration. We are trying to graft a search/problem-solving based approach to design onto a system which is based on the idea that design is exploration, not search. We have argued that design problems cannot be solved this way; they are too complex and there would be too many exceptions to any rules we might formulate. Rather it is necessary to explore the space of possible designs; to try things and observe their consequences.

If we look at what happens when the designer asks a ‘specialist’ for advice about some aspect of a design (typically “how do I solve this problem” where the ‘problem’ may be the selection of a material or a dimension, or some more general goal such as what type of structure to use or whether a building should be single storey or multi-storey), we find that the specialist uses the characteristics of the of the design description, often together with information solicited from the user, to select one of a number of possible materials or to compute an appropriate size for the component. However it does not (and cannot) investigate the wider implications of such a decision.

Assuming the specialist makes some attempt to ensure that the advice it offers is consistent with the current design solution there are two problems:

1. the number of possible sources of inconsistency the specialist can check is limited; and
2. even if it were possible for a specialist to ensure that its advice is *currently* consistent, if an inconsistency is subsequently detected what is the user to do?

These problems arise because the capabilities of specialists are finite. They are constrained to produce their advice based on the local problem context and cannot have knowledge of the consequences of their proposals for all design criteria in all situations.[8] Moreover, as we have seen, the resulting information cannot easily be truth maintained. We are therefore left with advice based on limited knowledge

[8] Newell (1990) terms this a *trap-state mechanism*: i.e., a mechanism which is both local (contained within a small region of the system) and whose result must be accepted. Such a mechanism “can itself (by assumption) only be a source of a finite amount of knowledge; and when that knowledge fails (as it must if the tasks are diverse enough) there is no recourse (also by assumption). Thus the system is trapped in a state of knowledge, *even if the knowledge exists elsewhere in the system*.” (p. 221, emphasis added) The point is that we are more likely to discover possible conflicts by utilising all the resources of the system rather than by relying on a single sub-system.

which may be invalidated if the designer violates any of the assumptions made by the specialist about the future development of the design solution. Even if a KS could detect and respond to the source of the inconsistency by backtracking to another solution, it is ultimately bound to fail. The problem is not solvable by search by definition, so no consistent solution can be found. The assumptions the specialist makes to justify its conclusions about the presence or absence of certain design characteristics either mark a failure to perform exploration (i.e., if a consistency test was performed they would be inconsistent) or if they are consistent now (an exhaustive consistency check was possible) they will become inconsistent.

When an inconsistency is detected what should the specialist do? The design description is simply a collection of constraints which are jointly inconsistent. How is the specialist to know which constraints it can change to restore consistency? The initial specification of a design problem is typically incomplete and inconsistent. These characteristics may or may not be apparent to the designer when design begins. More often the problem is discovered to be incomplete and inconsistent as the designer explores the space of possible designs. Even if the problem is initially consistent, it may become inconsistent as the designer completes the problems description by adding ‘missing’ requirements or as the designer makes decisions about the nature of the design solution. There is no reason to suppose that designers abandon such requirements or decisions when they discover them to be inconsistent (see, for example, (Rowe, 1987)). Nor is there any reason to suppose they should; it may be possible to achieve a much ‘better’ overall design solution, albeit to a slightly different problem, by retaining the designer’s (inconsistent) requirement or decision and relaxing one of the original problem requirements.

Even if we were to assume the ‘original’ or given set of (inconsistent) requirements were to take priority over the constraints defining a partial solution, i.e., those added by the designer, how does the specialist know which changes to the solution would be acceptable and which would not? To make such a decision would require at least a knowledge of the relative importance of the constraints and the possibility of their relaxation or the relative merits of various trade-offs between design the criteria. In short, *each* ‘specialist’ would have to know as much as the human designer. In fact it would have to know more, since much of this information is not available until the designer enters into a process of negotiation with the client or regulatory authorities responsible for the constraints. This does not seem to be a very promising basis for the design of ‘modular’ support systems.[9]

[9] Meyers, Pohl and Chapman (1991) in their work on the ICADS system have attempted to overcome this problem by centralising knowledge of possible conflicts between domain experts (i.e. specialists) and how these can be resolved in a single ‘Conflict Resolver’ module. While this allows domain experts to be developed independently of one another without worrying about inconsistencies in the advice they produce, it simply moves the problem elsewhere. With admirable candour they conclude that: “the assumption that the conflict resolution set required for the coordination of a representative number of domain experts can be largely predefined ... is obviously not valid.” and “the reasoning capabilities of the current ICADS working model ... cannot be extrapolated to a real world working model involving two or three dozen domain experts and a comparable number

If the specialist is sufficiently clever it will produce nothing since none of the alternatives it can propose are consistent. If we were to imagine an extremely sophisticated specialist that knew everything about the domain, i.e., it effectively completes the design in all possible ways in an attempt to find a consistent solution to the design problem it has been given, then if the requirements are inconsistent—as they typically will be—all it can do is fail. The useful information produced by such systems concerning the consequences of decisions and their inconsistencies with the problem requirements and other decisions is produced by accident. The answers they produce are useless *as answers to the designer's original question* as they are bound to be inconsistent, but the side-effects of trying to find the answers (i.e the exploration) is useful. Paradoxically the more limited the specialist, the more useful its results. As it becomes more limited, it makes more assumptions which result in more consequences being derived and hence more inconsistencies being discovered. Would it therefore not be better to produce this information in an orderly way—looking for the inconsistencies and recording them—rather than looking for something which doesn't exist?

If this argument is correct there is no point in the specialist even trying to be consistent since it can't know all the possible sources of inconsistency; and if it did it couldn't produce a solution. From a logical point of view all the solutions it can produce are equivalent; the specialist may as well claim the first thing that comes into its head. Resolving the problem requires either changing it or finding a new way to solve it.

9. The Exploration Knowledge Source

We are therefore investigating a third approach which we believe offers a more appropriate solution to these problems which involves allowing the system itself to explore the space of possible designs. We envisage a number of 'exploration' KSs either user-invoked or triggered by aspects of the current design description, whose function is to propose extensions to the current design description *without first attempting to determine if the extension is consistent*. Clearly the majority of such extensions will be inconsistent with the current design description. However this is not the point—they serve to highlight the implications of plausible design options. We are not suggesting that such exploration KSs should produce parameter values at random, rather they will investigate the consequences of appropriate values given the current design and the priority ordering on goals defined by the user. In fact such knowledge sources will be very like specialists, except that they simply serve to initiate a line of exploration rather than offering considered 'advice'. As such they can afford to be rather more permissive in their proposals.

of knowledge bases. Any attempt to predefine the necessary conflict resolution set, even if this were possible, would serve to constrain the design space and restrict rather than enhance the creative ambiance of the design environment." (p. 915)

We represent the results of such exploration as conditionals or counterfactuals:

$$\begin{aligned}&\mathit{conditionals}: \text{if } x = 5 \text{ then } y = 10 \ \text{ when } x \text{ is undefined}\\ &\mathit{counterfactuals}: \text{if } x = 5 \text{ then } y = 10 \ \text{ when, for example, } x = 2\end{aligned}$$

At present, while we can represent propositional conditionals, we do not derive them. The dependency information expressed by the conditional remains implicit in the justification structure and labels of the ATMS.

We can either allow the system to make assumptions or we can justify the hypothesis directly by the partial design description which triggered the 'exploration' KS. This greatly simplifies the representation of possible consequences, in that they are now a *monotonic* consequence of a partial design description. Conditionals summarise the justification structure, collapsing long chains of dependency relationships into single statements which summarise the structure of the problem. The (possibly counterfactual) conditional 'if $x = 5$ then $y = 10$' can be interpreted as stating that if the assumption '$x = 5$' were added to the current context (replacing any existing value for x) then '$y = 10$' would become derivable. By expressing such dependency information in this way we can highlight the implications of adding the assumption '$x = 5$' to the current context and suppress the fact that the conclusion '$y = 10$' depends on additional assumptions which also form part of the current context. The conditional is therefore only true in those contexts containing the additional assumptions on which it depends. However this is true of all derived information. The fact that the conditional depends on additional assumptions is recorded by the ATMS in the justification structure and the label of the conditional. If the underlying assumptions on which the conditional relies are subsequently discovered to be inconsistent, the conditional will lose support. Similarly it will cease to be true in any inconsistent extension of the current context.

By using the other knowledge sources to derive the consequences of possible design decisions we escape the problems of 'compartmentalisation' identified above and it allows us to use our existing blackboard architecture to derive the consequences of possible extensions to the current design description. Furthermore the 'exploration' knowledge sources seem to approximate more closely human design knowledge expressed as heuristics or rules of thumb about what might be a good idea to try in the current situation. An heuristic only says that if the condition is true the inference *may* be sound. The condition of an heuristic does rely on characteristics of the design description to identify those situations in which it might be useful from those in which they may not, but these do not attempt to be exhaustive. If the condition of an heuristic is true in some situation it does not mean that the heuristic will work; rather the condition only has to be sufficiently discriminating that the heuristic works enough of the time to be useful. If the heuristic doesn't work, simply try something else.

By relieving the SPSSs of the responsibility of knowing all the likely consequences of their suggestions (and in particular their unanticipated interaction with other parts of the design) we make it easier to engineer such capabilities in our systems. The combination of generator and test (together with the ability to summarise the resulting justifications structure as a conditional) works in the same way as a human designer, drawing on the full capabilities of the system in the form of KSs and module classes to assess the implications of proposals suggested by purely local considerations. To put it another way, a specialist no longer has to know everything. Rather than the specialist having to make explicit all the assumptions on which its advice is based—an impossible task as the set of such assumptions is infinite—it simply has to produce reasonable suggestions. For example, knowledge of statutory controls need no longer be encoded within the specialist (although it may be prudent to do so). While changes in legislation may invalidate a particular proposal made by the specialist, so long as this fact is detected by another KS the result will still be sound. The conditionals produced by the 'exploration' KSs cannot be used to answer precise questions about how a particular design problem *should* be solved, however we would argue such questions are unanswerable. The result of such exploration is design knowledge—knowledge of the type the designer is seeking about the structure of the problem.[10] One can imagine a persuasive front-end to the system responsible for presenting the results of the system's exploration to the user: "look, you can halve your costs if you use material x". We believe that such statements better capture the intuition underlying the notion of possible inference as it appears in the model of design support.

10. Conclusion

The task of a design support system has conventionally been conceived as one of providing the designer with solutions to specific parts of a design problem. The objective of such systems is to produce, either directly by elaborating the design description or indirectly in the form of 'advice' to the designer, consistent extensions to the current design proposal.

In this paper, we have argued that this goal is unattainable. It is important to stress that the problems identified above are not confined to EDS. In particular, they are not artefacts of the architecture of EDS. Rather the ATMS-blackboard makes explicit two important assumptions common to all design support systems, namely:

1. *Soundness*: that the advice given by the system should be 'good' advice. At a minimum it should be consistent with any existing partial solution.
2. *Constancy*: that the advice given should continue to be sound in the face of extensions to the design description.

While no system is infallible, we shall argue that a system is useful to the extent it approximates these goals. This can be seen most clearly if we examine what happens when a system violates one or other of these constraints. If a system fails,

[10] This is hardly surprising given that we have argued that the purpose of exploration is to produce such knowledge and (up to a point) it doesn't matter who does the exploration.

it is because the extension it proposes is (or becomes) inconsistent.[11] In many situations, although useless, this is relatively benign behaviour—the failure will be immediately obvious to the designer. However if we consider the designer's reasons for wanting to use such a system the failure of the system becomes more serious. If the designer is simply delegating a task to the system which the designer understands there is not much of a problem (although there is always the risk of an oversight on the part of the designer). However if the designer is relying on the system to augment his or her own skills and the advice produced by the system is inconsistent with the rest of the design the designer is stuck—they must adapt their design to the system's limitations or abandon the system and attempt to solve the problem on their own. On the other hand if the problem is more subtle and no obvious inconsistency results (for example if the designer and the system base their decisions on inconsistent assumptions which are never made fully explicit or if the system's advice subsequently becomes inconsistent) the consequence may be design failure.

To date these conditions have not been problematic because research efforts have mainly focussed on small, independent systems. The scope of such systems is typically fairly limited, so that it is not possible for the system to guarantee soundness. It is up to the designer to resolve any conflicts which arise between the advice offered by the system and that offered by other specialists or between the system and the existing design solution. Similarly, no attempt is made to ensure constancy. While the designer is free to re-run the system if an inconsistency arises or if the designer suspects the assumptions on which the system's advice rests have materially changed, the system itself cannot detect the revisions to the design description which would result in its advice being undermined. Independently, such systems are incapable of overcoming these limitations. Indeed it is only by concentrating on a single aspect of the problem that any progress has been possible in the development of such systems.

It is these assumptions of soundness and constancy which make the goal of modular 'assistants' capable of producing globally consistent solutions to parts of the design problem unattainable. The capabilities of such systems do not matter so long as their *purpose* is to propose consistent extensions of the design description. To solve any part of the design problem, a specialist effectively has to solve all of it, which eliminates all reasonable implementations. Design is essentially a holistic activity which cannot be broken down into sub-problems which are then solved independently.

We have proposed a solution to the problem of possible inference in the context of EDS based on the notion of 'exploration' knowledge sources. While this proposal is unique to the EDS architecture, we believe our general approach and in particular the objective of generating knowledge of the structure of the problem could be

[11] A system which claims to produce 'optimal' solutions can also fail if there exist better alternatives to the solution it proposes.

profitably applied to systems whose architecture is radically different from that of EDS. This is not just a change in emphasis, but a change in objectives which reflects an alternative view of the nature of the design process. We believe this to be a more fruitful approach than further attempts to develop existing search-based architectures.

Acknowledgements

The development of the Edinburgh Designer System was partially funded by the UK Science and Engineering Research Council as part of the Alvey large scale demonstrator project Design to Product (DtoP), other parts of which are funded by GEC plc and Lucas Diesel Systems (a division of Lucas Automotive Ltd.) whose active involvement in our research through the DtoP project we gratefully acknowledge, together with that of the other DtoP collaborators at Leeds University Department of Mechanical Engineering and at Loughborough University Department of Computing Studies, Department of Engineering Production, and the Human Factors in Advanced Technology research centre. Work on EDS is currently supported under SERC grant No. GR/F/6200.1.

Amaia Bernaras, Alistair Conkie, Ming Xi Tang and Nils Tomes read an earlier draft of this paper and made many helpful comments.

References

Buck, P., Clarke, B., Lloyd, G., Poulter, K., Smithers, T., Tang, M. X., Tomes, N., Floyd, C. & Hodgkin, E.: 1991, The Castlemaine project: development of an AI-based design support system, in *Artificial Intelligence in Design 91*, J. Gero, ed., Butterworth-Heinemann, 583–601.

Clark, K. L.: 1977, Negation as failure, in *Logic and Data Bases*, H. Gallaire & J. Minker, eds., Plenum Press, 293–322.

Dressler, O.: 1990, Problem solving with the NM-ATMS, in *Proceedings of the 9th European Conference on Artificial Intelligence*, L. C. Aiello, ed., Pitman Publishing, London, England, 253–258.

Gregory, S.: 1987, *Parallel Logic Programming in PARLOG*, Addison-Wesley.

Hayes-Roth, B.: 1985, A blackboard architecture for control, *Journal of Artificial Intelligence* 26, 251–321.

Junker, U.: 1989, A correct non-monotonic ATMS, in *Proceedings of the International Joint Conference on Artificial Intelligence*, Morgan Kaufman, 1049–1054.

de Kleer, J.: 1984, Choices without backtracking, in *Proceedings of the National Conference on Artifical Intelligence*, Austin, Texas, 79–85.

de Kleer, J.: 1986a, An assumption-based TMS, *Artificial Intelligence* 28, 127–162.

de Kleer, J.: 1986b, Problem-solving with the ATMS, *Artificial Intelligence* 28, 197–224.

de Kleer, J.: 1986c, Extending the ATMS, *Artificial Intelligence* 28, 163–196.

Kowalski, R.: 1979, *Logic for Problem Solving*, North Holland.

Lawson, B.: 1980, *How Designers Think*, Architectural Press, London.

Logan, B. S., Millington, K. & Smithers, T.: 1991, 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, 423–446.

Logan, B. S. & Smithers, T.: 1992, Creativity and design as exploration, in *Modeling Creativity and Knowledge-Based Creative Design*, J. S. Gero & M. L. Maher, eds., Lawrence Erlbaum, Hillsdale, New Jersey, 149–188, (in press).

Meyers, L., Pohl, J. & Chapman, A.: 1991, The ICADS expert design advisor: concepts and directions, in *Artificial Intelligence in Design 91*, J. Gero, ed., Butterworth-Heinemann, 897–920.

Newell, A.: 1990, *Unified Theories of Cognition*, Harvard University Press, Cambridge, Mass..

Nii, H. P.: Summer 1986, The blackboard model of problem solving and the evolution of blackboard architectures , *AI Magazine*.

Reiter, R.: 1980, A logic for default reasoning, *Artificial Intelligance* 13, 81–132.

Rowe, P. G.: 1987, *Design Thinking*, MIT Press, Cambridge, Mass..

Smithers, T.: 1987, The Alvey large scale demonstrator project Design to Product, in *Artificial Intelligence in Manufacturing, Key to Integration*, T. Bernhold, ed., North Holland, Amsterdam, 251–261.

Smithers, T., Conkie, A., Doheny, J., Logan, B., Millington, K. & Tang, M. X. : 1990, Design as intelligent behaviour: an AI in design research programme, *Artificial Intelligence in Engineering* 5, 78–109.

Smithers, T. & Troxell, W. O.: 1990, Design is intelligent behaviour, but what's the formalism, *Artificial Intelligence in Engineering Design, Analysis and Manufacturing* 2, 89–98.

Tsang, J. P.: 1991, A combined generative and patching approach to automate design by assembly, in *Artificial Intelligence in Design 91*, J. Gero, ed., Butterworth-Heinemann, 485–502.