Using Reason Maintenance Systems to Support Ill Structured Problem Solving

Brian Logan, David Corne

Department of Artificial Intelligence
University of Edinburgh
Forrest Hill
Edinburgh, UK

and

Tim Smithers

Artificial Intelligence Laboratory
Vrije Universiteit Brussel
Brussels, Belgium

Abstract

Problem solving systems incorporating a truth or reason maintenance component have been developed for a number of different domains, including hypothetical reasoning, diagnosis, planning and circuit design.[1] Almost without exception, the problems addressed by these systems are well structured, that is, there exists a complete and consistent statement of the problem requirements when problem solving begins. In this paper we argue that RMS (and in particular assumption-based truth maintenance systems) can also be applied to ill structured problems. We describe a hybrid ATMS-blackboard architecture which has been employed in the development of a number of design support systems in different domains and briefly outline some of the questions raised by our experience with the architecture.

DAI Research Paper No 596, August, 1992

This is a revised version of a paper presented at the ECAI'92 workshop on the Application of Reason Maintenance Systems, Vienna, Austria, 4th August, 1992

[1]An excellent review of the literature on RMS applications can be found in Martins (1990).

1 Introduction

A problem is ill structured if it is not well structured, that is, if there is no complete and consistent problem statement when problem solving begins (Simon, 1973). Many real world problems are ill structured. For example, in deciding whether to buy a house or a car or where to go on holiday, one's decision may be influenced by a wide range of factors many of which may only emerge when the various alternatives have been considered in detail, and initial requirements may have to be revised in response to unexpected difficulties or opportunities. In this paper we concentrate on a particular kind of ill structured problem, *design problems*. Design problems are interesting because part of the problem definition is given in the form of requirements the designed artefact or process must meet. However these requirements are typically incomplete and/or inconsistent and considerable effort is often required to formulate the problem.

It is rare for any part of a designed artefact to serve only one purpose, and it is frequently necessary to devise a solution which satisfies a whole range of different requirements. In many cases the stated objectives are in direct conflict with one another and the designer cannot satisfy one requirement without causing problems elsewhere. Different trade-offs between the criteria result in a whole range of acceptable solutions, each likely to prove more or less satisfactory in different ways to different clients and users. The value judgments regarding 'trade-offs' between criteria are context dependent, and the balance of satisfaction for such requirements are often unclear until the designer explores the various possibilities in appropriate detail. Such value judgments apply not only to the 'qualitative' criteria such as aesthetics, but also to the relative importance of quantitative criteria which themselves may be susceptible to objective measurement. Questions about which are the most important problems and what kinds of solution most successfully solve these problems are also value laden, and the answers given by designers to these questions are therefore frequently subjective and highly context dependent.

As a result, a large part of the design process is devoted to discovering the nature and scope of the task set by the requirement description. Particular aspects of the problem may suggest certain features of solutions, but these solutions in turn create new and different problems. 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 structuring of relationships between these criteria that forms the basis for the design process (Lawson, 1980). The fundamental objective is therefore that of understanding the structure of the problem and analysing the inter-relationships between criteria to gain some insight into the relationship between any individual design decision and all of the other decisions which together define the solution.

2 Solving Ill Structured Problems

In attempting to solve such problems designers explore the space of possible solutions trying out decisions and investigating their consequences in a way which develops both the solution and the problem requirements (Smithers & Troxell, 1990). There is no meaningful distinction between the analysis of problems and the synthesis of solutions in this process; problems and solutions are seen as emerging together rather than one logically following from the other. The problem is explored through a series of attempts to create solutions and understand their implications in terms of other criteria. The designer comes to understand the critical relationships and possible forms as a solution evolves. Between generic solutions design is less a search for the best solution than an exploration of the compromises that give sufficient solutions. 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.

This process of exploration frequently results in inconsistencies between the designer's best guess at a solution and the problem requirements or between the current decision and the existing partial solution. In some cases such inconsistencies can be resolved by modifying the proposed solution. However, if no solution which meets the problem requirements can be found, then the problem requirements are themselves inconsistent (relative to the knowledge of the designer) and the problem is insoluble. To 'solve' such a problem it is necessary to redefine it. For example, the problem of designing a house with a floor area of at least $100\,\mathrm{m}^2$ costing less than £10,000 is probably insoluble given existing construction techniques, prevailing statutory requirements etc. without some redefinition of the concept of 'house'. When an inconsistency arises, it is unrealistic to expect the designer to restore consistency immediately. Indeed it is often only be exploring the implications of the conflicting design decisions and requirements that a choice can be made about which decision to abandon or which requirement to relax and in many cases considerable work is required before the conflict can be resolved. In such situations we are really working with several inconsistent but related partial designs.

Previous attempts at supporting this process using computers have tended to adopt one of two approaches: deriving the characteristics of a design given a description of the solution; and generating a solution or part of a solution given the design goals and requirements. Such systems can be effective when the problem and solution are well defined. However they are typically incapable of operating with incomplete or inconsistent information. If the problem requirements are incomplete or inconsistent then no consistent solution can be found. Similarly, if the proposed solution is incomplete or inconsistent no consistent evaluation is possible and the question of whether the solution meets its design goals is meaningless. The response of most systems to these problems is to ignore them. Existing CAD tools are ill suited to the exploration of ill structured problems, tending to focus on individual design characteristics such as cost, structural stability etc. With no overall framework to integrate the individual tools, the problem of inconsistencies between criteria is not addressed and they provide no support for the process of exploration or problem redefinition. The designer is left with the task of reformulating the problem requirements and/or the solution and trying again.

3 The Edinburgh Designer System

In an attempt to overcome these difficulties we have implemented a design support system, the Edinburgh Designer System (EDS), which allows the designer to explore the implications of inconsistent sets of assumptions (Logan, Millington & Smithers, 1991; Smithers et al., 1990; Smithers et al., 1992). EDS doesn't actually design anything—rather it attempts to support the designer in exploring the space of possible designs.

EDS consists of four principal subsystems: *knowledge representation*; *inference*; *consistency maintenance*; and *context management*.

3.1 Knowledge Representation

In EDS, domain and design knowledge is represented as a series of *module classes*, related by *part_of* and *kind_of* relations. Domain knowledge expresses facts about the domain, such as the heat conduction properties of different building materials and the properties and attributes of basic objects and processes associated with the domain—its ontology—such as walls, windows, doors, etc. It also reflects scientific, technological, sociological and statutory knowledge and constraints relevant to the domain. This knowledge is represented declaratively in the form of constraints and is organised hierarchically using *specialisation* (kind_of) relationships between representation objects. Design knowledge derives from previous designs and is concerned with how domain knowledge is used to define and solve design problems; how the space of possible designs is explored and how a developing design problem structure is created, modified, and refined. Design knowledge includes useful decomposition criteria and strategies, synthesis and analysis methods and techniques, and required validation, documentation, and presentation procedures, for example. While domain knowledge governs the underlying behaviour of a design problem, design knowledge is used in deciding how it is to be configured—what priority to put on requirements and constraints, what is to be ignored, and what is to be included etc. Since, during the design process, this problem structure is often modified and revised as its nature is explored and understood, design knowledge must also be represented declaratively. In our representation scheme an *aggregation* (part_of) relationship is used to 'chunk' design knowledge expressed in terms of domain knowledge.

Each module class declares a set of parameters, variables and constraints which define a particular class or type of object. For example, in the house design problem mentioned above, we would typically have module classes containing information about houses, rooms, walls etc. Relevant parameters for a *house* module class might include the floor area of the house A, the total cost of the house C and the cost/m^2 floor area U, together with constraints expressing dependencies between these parameters, such as $A \times U = C$. Instances of these module classes form the *design description* which contains the requirements, domain knowledge and design decisions relating to a particular problem.

3.2 Inference

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.[2] The system attempts to assist the designer in exploring the consequences of 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 tries, where possible, to 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.

There are currently four main support systems within EDS (Smithers et al., 1990):

1. the *Evaluation Engine* which handles value propagation, constraint satisfaction and expression simplification;

2. the *Algebraic Manipulation Engine* 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* which performs value interpolation and relational operations on tabular data; and

4. the *Spatial Reasoning Engine* which performs spatial inferencing.

Control of the interactions between the support systems is in the style of a blackboard system (Hayes-Roth, 1985). 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. At each blackboard cycle the system applies the collection of (user defined) knowledge sources to the design description represented on the blackboard, reviewing any uncompleted work in the light of what is discovered and adjusting its inference priorities accordingly. The designer 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 designer's decisions, while still giving priority to user input.

3.3 Consistency Maintenance

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. One approach to this problem would have been to use a justification-based truth maintenance system (Doyle, 1979). However while such systems guarantee

[2] In general the support systems derive necessary consequences of the designer's decisions. For a discussion of the problems of integrating the derivation of possible consequences into a monotonic ATMS see (Logan, Corne & Smithers, 1992).

consistency, they do so by forcing the user to maintain a single consistent design description. All a contradiction between two 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. Moreover, because the TMS algorithms allow only one solution to be considered at any one time, it is difficult to compare the implications of alternative design decisions.

If a design support system is to effectively support the exploration activities of the designer, the various incompatible design alternatives must be considered in parallel. This, together with the need to derive as much information as possible from inconsistent design descriptions, led to the adoption of an assumption-based reason maintenance system (ATMS) for EDS (de Kleer, 1986a). Each item on the blackboard (including the knowledge source activation records in the agenda) is associated with an ATMS node. In EDS, the assumptions represent the problem requirements, knowledge of the domain and the basic design decisions made by the designer.

The ATMS forms the core of the system and all of the other system components are implemented using the facilities it provides. 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. The justification structure records the dependencies between design decisions about problem requirements and solutions and their consequences.

3.4 Context Management

While the ATMS is effective in restricting inferences to those based on consistent premises, in general the consistency of a set of assumptions is too weak a criterion to determine the relevance of a possible inference and typically results in the derivation of a large amount of redundant information. To provide a degree of control the user can partition the information on the blackboard into a number of distinct contexts or *views* each of which contains the information relating to part of a particular design solution or task using an assumption based context management system (ACMS).[3]

A context is any (possibly inconsistent) set of assumptions.[4] A *view* is a named context, i.e. a named set of assumptions. The assumptions are termed the *assumption base* of the view. The *extension* of a view is everything that can be derived from its assumption base. More precisely, an item is a member of a view if the assumption base of the view subsumes one or more of the environments in the item's label. Views are defined relative to the the set of all assumptions or 'universal assumption base', $\mathcal{V}$. The assumptions forming a view's assumption base are specified by its *defining abstraction.* The definition of a view can be either intensional or extensional depending

[3] Views represent an extension of the *focus environments* described by Forbus and De Kleer (1988), except that there can be more than one view and the assumptions defining a view need not be consistent. In the terminology of Martins and Shapiro (1986) a view is a *belief set.*

[4] Note that this differs from de Kleer's terminology (de Kleer, 1984), where *context* is taken to mean the set of data derivable from an environment. Our use of the term is closer to that of Martins and Shapiro (1986).

on whether its defining abstraction is open or closed. An intensional definition is a predication P defining a set of assumptions $\{x : Px\}$ where x ranges over $\mathcal{V}$ and P is a boolean test defined on node labels or their contents. A small number of system-defined tests are provided by the ACMS. More complex view definitions can be constructed from these primitive definitions using the set operators union, intersection and set-difference. An extensional definition is simply a list of assumptions $\{a_1, \ldots, a_n\}$. Unlike an open abstract which produces different results at different times, a closed abstract always returns the same set of assumptions. The *current* assumption base of a view is found by evaluating its defining abstraction relative to the universal assumption base. The evaluation of an abstract returns the assumptions for which it is currently true. Evaluating a closed abstract simply returns the corresponding list of assumptions Evaluation effectively freezes the definition of a view relative to the current state of the DDD. Assumptions or inferences introduced into the DDD after the evaluation of the view's defining abstraction do not form part of the view's definition.[5]

The set of views forms a tree. Each view is associated with a unique node in the tree. A 'view-name' is a sequence of named abstractions of the form

$$name \cdot view\text{-}name$$

Within this structure there are two distinguished views called '*univ*' (the universal or 'root' view) and '*nil*' (the empty view) corresponding to the set of all assumptions $\mathcal{V}$ and the empty set $\mathcal{E}$ respectively. The name of a view specifies its location within the tree relative to the root node. For example

$$univ \cdot widget$$

denotes the the view *widget* which is a child of the universal view. As a syntactic convenience, any view name beginning with the view name separator '·' is assumed to be an absolute view name. For example

$$\cdot assembly \cdot sub\text{-}assembly \cdot component\text{-}1$$

names the view *component-1* which is a child of the view *·assembly·sub-assembly*. All assumptions are by definition members of the universal view and all views are defined relative to it.

The organisation of views into a tree allows 'relative naming' of views. For example, the components of an assembly can have the same name in different assemblies. To refer to the same part in *assembly-1* and *assembly-2* we could write

$$\cdot assembly\text{-}1 \cdot widget$$

and

$$\cdot assembly\text{-}2 \cdot widget$$

The view structure can be thought of as a task decomposition for the design problem —keeping track of the various alternative design proposals and the information required

[5] Note that assumptions which are subsequently discovered to be inconsistent with the definition of the view will not be removed from the assumption base, although the ATMS will partition environments to maintain consistency.

to solve particular problems, both as a record of the design process (which alternatives have been tried and what was learned) and to allow the designer to partition the problem appropriately (Logan, 1989). The views system provides a level of organisation above that of the dependencies maintained by the ATMS which reflects the designer's interests and goals. Whereas justifications model the fine structure of the problem, views model the broad structure.

Note that the definition of a view is independent of its position in the view structure. The name of a view simply provides a convenient way to refer to its defining abstraction. Since a view name ultimately reduces to the abstraction defining its assumption base, it is straightforward to define a view in terms of other views. This provides a primitive form of inheritance between views. As the contents of a view change so too do those views defined in terms of it. Defining views in terms of other views allows the construction of complex inheritance structures. While the view structure is a tree, the inheritance structure is a directed acyclic graph. Inheritance can sometimes be a problem however, for example, when we want to create variants of an existing design, and to avoid this the user can force the evaluation of part or all of a defining abstraction to obtain its current extension which is stored as the view's definition. This has the effect of 'copying' the definition of the view and breaking any inheritance links it has to other views.

The system provides a number of other primitive operations on views, including copy and deletion. Copying the view $view_1$ copies the sub-tree of views rooted at the view $view_1$ to another view, $view_2$, resolving all references to views below $view_1$. References to views outwith the tree rooted at $view_1$ are preserved; references to views below $view_1$ (and the corresponding definitions) are copied to the new tree below $view_2$. When combined with evaluation and/or redefinition, *copy* can be used to efficiently explore the implications of alternative parameter values. Deleting a view deletes the view and its children (if any) from the view structure. Any view whose defining abstraction references a deleted view is marked as undefined but is otherwise unchanged. In particular, it continues to reference the now non-existent view. Any view defined in terms of an undefined view is also marked as undefined. Together with *copy*, evaluation and redefinition, this allows the rapid construction and reorganisation of the task structure in response to the designer's exploration of the problem. For example, the designer can delete the subtree of views representing part of a solution and replace it with an alternative solution by substituting a copy of another part.

At any time one or more views are selected as the user's current focus of attention. Only those items which are members of the current view(s) are visible to the knowledge sources and hence can form the basis of further inferences. Potential inferences which could be performed in other views—for example in views which inherit from the the current view—are ignored. Assumptions made within a current view are automatically added to its defining abstraction and hence to its assumption base. This may in turn result in new inferences being made or previously derived inferences gaining support within the view.[6] Switching views can therefore be used as a simple but effective means of process control. By changing contexts, the user can focus the system's attention on a particular part of the problem or on a particular kind of inference. As different assumptions become visible, so different knowledge sources are activated and different kinds of inferences performed.

[6] Note that while inference is confined to the set of current views, the operation of the ATMS is not.

When a view becomes current (i.e. when it becomes a current task), the system must bring it up to date by performing any inferences made possible by information (assumptions or inferences) added to the DDD since the view was last visited. The defining abstraction of the view is re-evaluated and the resulting assumption base is used to rebuild the view's extension. If the view is defined in terms of other views, their defining abstractions must first be evaluated and so on recursively. This effectively recomputes the inheritance relation between the current view and the views forming its definition. New bids are posted for any work based either partially or wholly on 'new' information. Any pending KSARs which were not processed the last time the view was visited (and which have not subsequently been executed in other views) are also added to the agenda. The system goes to some lengths to minimise the cost of switching views by maintaining a record of when each view's defining abstraction was last evaluated, and by attempting to ensure that the KSARs produced on switching views have not been processed before. This recognises that out of any collection of contexts or variants, many are effectively 'dead' in that they will never be considered again. As the extension of a view is rebuilt only when the view is visited, the overhead of a view is limited to the small amount of memory required to store the view's definition. Given the relatively low cost of creating views and the fact that each assumption is stored only once there is little penalty in creating as many views as necessary to solve a particular problem.

4 Supporting the Design Process

Design proceeds by creating instances of module classes and assigning values to their parameters to define one or more possible solutions.[7] When the user makes an assumption in one of the current views, one or more datum nodes are created to hold the new information. For example, in attempting to solve the house design problem, the designer might begin by creating an instance of the *house* module and assigning values to some of its parameters: $A \geq 100$, $C \leq 10,000$ etc. 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 (KSAR) which is scored and merged into an agenda. At each blackboard cycle, the KSAR with the highest score is executed and the results are added to the design description. As the design proceeds the consequences of the designer's assumptions are derived by the support systems. Such derived information typically relates to the predicted performance of the designed artefact (including any constraints violated by the proposed design) and any parameter values which can be inferred from the designer's assumptions or their consequences and the constraints linking these values. In the example above, the assumptions made by the user together with the constraint $A \times U = C$, can be used by a *valuePropagation* knowledge source to infer the additional constraint $U \leq 100$. 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.

In general, the support systems are triggered automatically by changes in the design

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

description. 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: 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 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. However this approach is not practical in its pure form. While the knowledge sources 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, 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. In practice, the system uses a simple scoring policy for KSARs which embodies a number of simple heuristics applicable in a range of domains. 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).

As new values for parameters or bounds on them are assumed or derived, consistency checks are performed between constraints and values by the *valueConflict* knowledge source. 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 cause the ATMS to partition the assumptions into mutually consistent sets. An inference which is only derivable from inconsistent assumptions loses support and cannot form the basis of further inferences. If there is no conflict, EDS marks this by justifying the datum $\langle$*consistent*$\rangle$ and proceeds as usual. For example, we may know that the lower limit on construction costs is £110/m^2. Adding the constraint $U > 110$ to the problem description leads to a justification for $\langle$*false*$\rangle$, recording the fact that the three assumptions $A \geq 100$, $C \leq 10,000$ and $U > 110$ are jointly inconsistent (and hence that the problem is insoluble) and invalidating any inferences based on these assumptions. However the constraint is consistent with each of the problem requirements individually and it can be used to derive new bounds on the values of A and C, i.e., that $A \leq 90.9$ and $C \geq 11,000$. Even if if the design fails to violate any constraints the designer may elect to pursue several different designs in parallel in an attempt to determine which gives the best overall performance, or to determine the sensitivity of the derived performance to the values of the design parameters. This will typically involve trying several alternative values for parameters until the constraints are satisfied or the relative performance of the various alternatives is understood.

Note that the presence of inconsistencies in the design description does *not* imply that the proposed design is unsatisfactory. Rather it simply means the designer's assumptions are jointly inconsistent. The evaluation of a design solution is relative to other designs both existing and entertained and what the designer believes to be possible. An 'objective' measure of the performance of the design such as its cost is value free; it only becomes meaningful relative to the cost of other similar designs or to the problem requirements (which themselves express an expectation about what constitutes a 'good' solution). The same design can be both good and bad from different points of view. For example, a particular house design may be expensive in comparison with other houses of a similar area and type, but may represent good value when the site conditions and the level of finishes is taken into account. Putting gold taps in the bathroom not only increases the cost of the house, it changes the way the cost should be evaluated.

Different members of the design team will evaluate the same design in different ways, each emphasising different characteristics of the solution. Indeed, individual designers will evaluate the same design differently at different times as they achieve a better understanding of what is achievable using the available technology. A design which was previously thought to be mediocre may acquire a higher score if it discovered that the alternatives are even worse. Similarly, a design which represents a 'good compromise' may be abandoned when a constraint is relaxed.

When an inconsistency arises or the design is found to be unsatisfactory (as is typically the case), the designer has several options. The designer can ignore the inconsistency and continue to pursue the development of the inconsistent design based on what can be coherently derived. The opportunistic nature of the blackboard control strategy means that whatever can be consistently derived within the current view will be derived. Multiple assignments to parameters (giving rise to multiple alternative solutions) and their interactions are handled automatically by the ATMS as are inconsistencies between parameter values and any assumed constraints. This approach may be appropriate when, for example, the inconsistency is considered minor or peripheral and the main interest lies with the consequences of some (consistent) set of assumptions which are considered central to the proposed solution.

Alternatively, the designer can attempt to eliminate the inconsistency by modifying the offending parameter values using information on which assumptions are jointly inconsistent provided by the ATMS. This may involve assigning new values to parameters or relaxing constraints, i.e. changing the requirements or adopting a different approach. To assist the designer in understanding the dependencies between assumptions and derived results, EDS provides various utilities which allow the user to examine the contents of the DDD. For example, using the *graphical explanation* facility, the user can display a graphical representation of how a parameter was inferred by viewing its *justification*, *environment* and *inferencing method*. In particular the user can discover the reason for the inconsistency signaled above by requesting that the system display the mutually inconsistent assumptions (and their consequences) which led to the derivation of $\langle \mathit{false} \rangle$.

However, if there are several different ways of ‘patching’ the design to restore consistency which are to be investigated in parallel, it may be more appropriate to create a view for each of the alternative solutions. For example, in determining how to overcome the problem identified above, the designer may make further assumptions to explore the implications of relaxing one or both of the original problem requirements, for example that $A = 90$ or that $C = 11,000$. These assumptions are inconsistent with the existing design solution and with each other. However the inconsistencies between the assignments are in a sense irrelevant. The designer does not care that they are inconsistent and the system should not pursue inferences based on their union.[8] While these inconsistencies will be trapped by the ATMS, they are of no interest to the designer.[9] By placing assumptions relating to different designs in different views, the designer can avoid the derivation of such meaningless inconsistencies and partition the design description in response to their exploration of the problem. Such variant views can either be used as a ‘scratchpad’ for rough working, merging the results back into the main view structure by redefining the part view as the chosen variant or they can form part of the final design record, in which case the component view can be redefined to inherit from the appropriate part view.[10]

This approach has a number of advantages. The designer can pursue the development of the design without worrying about inconsistencies except insofar as they indicate the shortcomings of a proposed solution. The ATMS ensures that only valid inferences are drawn from the current design description by automatically detecting inconsistent sets of assumptions and violated constraints and partitioning the design description to restore consistency. Moreover the opportunistic nature of the blackboard control strategy means that whatever can be consistently derived from the design description will be derived. However, it does change the relationship between the user and the ATMS.

5 Reason Maintenance in Ill Structured Problem Solving

In a conventional RMS based problem solving system, the RMS is usually seen as an independent module associated with the problem solver (de Kleer, 1986b). The problem solver communicates the results of its inferences to the RMS whose task is to keep a record of the dependencies between propositions and use these dependencies to inform the problem solver of which propositions it should believe. In a support system there is no ‘problem solver’ as such. In design, both the problem and the problem-solving process are ill-structured. The system has no well-defined goal such as ‘find a configuration which satisfies the given constraints’. Rather it looks for ‘interesting’ consequences of the assumptions about requirements and parameter values made by the

[8] Whether two sets of assumptions should be considered disjoint in this sense is itself dependent on the current context. At some point in the future the designer may wish to consider amalgamating these two alternatives, at which point their consistency or otherwise does become an issue.

[9] Note that without the ATMS, these assumptions would result in the derivation of an infinite sequence of values for A and C.

[10] The problem of when a modification or revision should entail the creation of a new view is left to the designer. The views system is intended to facilitate the exploration of alternative solutions and any predefined strategy would simply constrain the exploration process. This is discussed in more detail in the next section.

user. Inconsistencies are particularly interesting both because they terminate a line of exploration (from the system's point of view), and also because they are interesting in their own right—they indicate problems with the requirements or solution.

These assumptions and their consequences record explorations of the space of possible designs and are not necessarily solutions to the same problem. As a result interpretations are not meaningful. In a conventional RMS-based problem-solving system, a solution is implicitly identified with the notion of a 'maximal environment'. In EDS, the design description contains both decisions from alternative approaches to the problem and assumptions about alternative requirements and constraints. As a result, the efficiency of the ATMS is perhaps not as important as it is in more conventional systems.[11] With appropriate use of views to segregate inconsistent designs and control the derivation and propagation of inconsistencies, few extraneous nogoods are created and we have not found it necessary to limit label propagation to the current view (see for example (Dressler & Farquhar, 1990)). While this would be advantageous in some situations, label propagation outwith the current view does not seem to be a significant overhead in the domains we have addressed.

Moreover the relationship between the support systems and the RMS is rather different, in that we are interested not only in whether sets of assumptions are consistent or not, but in the justification structure itself. From the system's point of view, the construction of the justification structure is not a means to an end, but (almost) an end in itself. It records the results of the system's exploration of the user's decisions and represents the structure of the problem. The justification structure records the dependencies between criteria that the designer is trying to understand and forms the basis of further exploration. This results in a major user interface problem which does not arise in more conventional RMS based problem solving systems (or at least not to the same extent) which are primarily interested in the answer rather than the structure of the problem—that of presenting the justification structure to the user in an intelligible way. Although the system provides a number of graphical tools for examining the justification structure, there is no easy way for the user to identify an assumption or datum node (for example in defining a view) without reference to the node-tag used by the ATMS to to refer to the node in environments and justifications. The ATMS simply maintains dependencies between data items—it has no knowledge of the contents of the nodes or assumptions. It does not make sense to talk about "*the* value of parameter p of instance i" as p may have several values.

6 Extending the Architecture

EDS has been used extensively by several of the collaborators in the Alvey large scale demonstrator project 'Design to Product' (DtoP) which focussed on the design of light electro-mechanical components and formed part of the final DtoP demonstration system (The DtoP Consortium, 1991). The architecture has also been applied in a number of other domains including drug design (Smithers et al., 1992), studies of of nuclear power

[11]The ATMS used by EDS is not particularly efficient; there are a number of possible optimisations which we have never bothered to make because the overhead of the ATMS within the system as a whole is relatively small. A much bigger problem is controlling the knowledge sources (Smithers, 1989).

systems design (Furuta & Smithers, 1991) and option trading. Our initial experience with the architecture has been encouraging. The ATMS-blackboard and ACMS have proved useful in supporting *ad hoc* explorations of the trade offs between design criteria in these domains. More importantly, our initial experience has served to highlight some of the shortcomings of the model of design support which the system embodies and we briefly summarise these below. While in the main these are not problems with the ATMS-blackboard as such, it seems likely that significant revisions to the architecture of the ATMS-blackboard will be necessary if they are to be overcome.

Further development of the architecture is limited by our understanding of the design process. A major problem with the current system is that it has no understanding of the design task and consequently which inferences it should perform in a given situation. The system lacks any concept of the importance of a failure to achieve a particular design goal. There is no way in the current scheme to distinguish between what might be termed trivial inconsistencies due to minor differences in parameters values and the 'radical inconsistencies' which are, at least initially, of greater interest to the designer, and which may completely invalidate an approach to the problem. At present if a set of assumptions prove to be inconsistent no further inferences are possible in any environment subsumed by these assumptions. This is untenable if the knowledge sources are interpreted as implementing different logics.

One solution to this problem is to include (a name for) the knowledge source itself in the antecedents of a justification. This information is currently available in the method 'slot' of the justification in the ATMS. Making explicit the dependency of a derivation on the knowledge source responsible for the inference simplifies the subsequent truth maintenance and allows the system to distinguish between different kinds of inconsistencies. For example, it becomes possible to distinguish between 'rounding errors' in the calculation of the same parameter by different methods and 'semantic' inconsistencies arising as a result of conflicting design decisions.[12]

This in turn suggests that the knowledge sources themselves may profitably be viewed as first-class objects maintained by the ATMS. This facilitates reasoning about the knowledge sources for control purposes and renders them subject to both consistency maintenance and context management, by allowing the user to specify which knowledge sources should form part of the current task definition. For example, the system can identify those knowledge sources which are mutually 'inconsistent', (in that they will always derive inconsistent results from the same assumptions) or those which are considered appropriate in a particular context. A logical extension of this would be to allow the system to reason about views as first-class objects—for example to infer from the current task what the current view should be. Such a self-referential system offers a significant increase in expressiveness, in allowing statements (and hence reasoning about) other statements or collections of statements, such as their utility in particular contexts or the completeness and consistency of design descriptions. It also provides a natural interpretation of knowledge sources as procedurally interpreted views.

In an attempt to overcome these problems we are currently implementing a series

[12] At present EDS handles rounding errors using an 'epsilon' value which is the same in all contexts, as without knowledge of the justification structure, the knowledge source responsible for detecting value conflicts cannot determine how the parameters were derived.

of extensions to the ATMS-blackboard architecture including the introduction of user-defined preference orderings over problem requirements, knowledge sources and views and the representation of knowledge sources and views as first class objects within the ATMS-blackboard. We believe such extensions can form a framework for a more effective control of inference.

7 Conclusions

We have argued that RMS can form an appropriate framework for solving ill structured problems. We have described a design support system, the Edinburgh Designer System and its ATMS-blackboard architecture, and illustrated how it supports the exploration of design problems. The implications of this approach for the relationship between the user of the system and the ATMS have been briefly discussed and some of the problems which result—problems which are not often found in 'conventional' RMS applications: problem redefinition and interpretation construction; the interpretation of inconsistencies; and the presentation of the justification structure—outlined. Based on our experience with the system, we have argued that to improve the support provided by EDS, the ATMS-blackboard architecture must be extended and we have briefly outlined some of our current work which aims to improve the system's understanding of the task the designer is currently pursuing.

We believe this approach is applicable to decision support systems and ill structured problem-solving in general, and that such systems offer a fruitful new area for the application of reason maintenance systems.

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. The application of the EDS architecture to drug design was funded by the joint SERC/DTI Information Technology programme under grant number GR/F/3567.8 in collaboration with Logica Cambridge, British Biotechnology Limited and CamAxys Limited. Work on EDS is currently supported under SERC grant No. GR/F/6200.1.

References

Doyle, J. (1979). "A Truth Maintenance System", *Artificial Intelligence* 12, 231–272.

Dressler, O. & Farquhar, A. (1990). "Putting the Problem Solver Back in the Driver's Seat: Contextual Control of the ATMS", in *ECAI-90 Workshop on Truth Maintenance Systems*.

Forbus, K. D. & Kleer, J. de (1988). "Focusing the ATMS", in *Proceedings of the Seventh National Conference on Artificial Intelligence*, American Association for Artificial Intelligence, 193–198.

Furuta, K. & Smithers, T. (1991). "Numerical Methods in AI-Based Design Systems" , in *Applications of artificial intelligence in engineering VI, Proceedings of the 6th International Conference on Applications of Artificial Intelligence in Engineering*, G. Rzevski & R. A. Adey, eds., Computational Mechanics Publications, Southampton, 45–58.

Hayes-Roth, B. (1985). "A Blackboard Architecture for Control", *Journal of Artificial Intelligence* 26, 251–321.

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

de Kleer, J. (1986aa). "An Assumption-based TMS", *Artificial Intelligence* 28, 127–162.

de Kleer, J. (1986bb). "Problem-solving with the ATMS", *Artificial Intelligence* 28, 197–224.

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

Logan, B. S. (1989). "Conceptualizing Design Knowledge", *Design Studies* 10, 188–195.

Logan, B. S., Corne, D. W. & Smithers, T. (1992). "Enduring support: on defeasible reasoning in design support systems", in *Artificial Intelligence in Design '92*, J. S. Gero, ed., Kluwer Academic Publishers, Dordrecht, 433–454, (in press).

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.

Martins, J. P. (1990). "The Truth, the Whole Truth, and Nothing But the Truth: An Indexed Bibliography to the Literature of Truth Maintenance Systems", *AI Magazine*.

Martins, J. P. & Shapiro, S. C. (1986). "Theoretical Foundations for Belief Revision", in *Theoretical Aspects of Reasoning about Knowledge*, Joseph Y. Halpern, ed., Proceedings of the 1986 Conference, Morgan Kaufmann, Los Altos, 383–398, Monterey, March 1986.

Simon, H. A. (1973). "The Structure of Ill Structured Problems", *Artificial Intelligence* 4, 181–201.

Smithers, T. (1989). "Intelligent Control in AI-Based Design Support Systems", Department of Artificial Intelligence, Edinburgh University, Techincal Report No. 423, Prepared for the Workshop on Research Directions for Artificial Intelligence in Design, Stanford University, March 1989..

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., Tang, M. X., Tomes, N., Buck, P., Clarke, B., Lloyd, G., Poulter, K., Floyd, C. D. & Hodgkin, E. E. (1992). "Development of a knowledge based design support system", *Knowledge Based Systems*, (in press).

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.

The DtoP Consortium (1991). *'Design to Product' An Alvey Programme Large-Scale Demonstrator Project: Final Report to the Department of Trade and Industry and the Science and Engineering Research Council on Contract Number LD/004*, (Commercial in Confidence).