Group Decision and Negotiation, 2:259–278 (1993)

Supporting Conflict Management in Cooperative Design Teams

MARK KLEIN
mklein@atc.boeing.com
Boeing Computer Services, PO Box 24346, MS 7L-64, Seattle, Washington 98124-0346

Abstract

The design of complex artifacts has increasingly become a cooperative process, with the detection and resolution of conflicts between design agents playing a central role. Effective tools for supporting conflict management, however, are still lacking. This article describes DCSS (the Design Collaboration Support System), a system developed to meet this challenge in teams with both human and machine-based design agents. Every agent has an "assistant" that provides domain-independent conflict detection, explanation, and resolution expertise. Agents and assistants cooperate synergistically during design to resolve conflicts as they arise. Agents describe their design decisions using a generic least-commitment design model. Assistants provide libraries of conflict detection techniques, built on unsatisfiable constraint set detection, to help flag conflicts between decisions made by different agents. Once a conflict has been detected, assistants use design rationale provided by the agents as well as a generic conflict-cause taxonomy to suggest possible explanations for the conflict. Finally, assistants provide suggestions for resolving a conflict by instantiating domain-independent strategies using domain-specific expertise elicited from the design agents. DCSS has been used successfully to support cooperative design of Local Area Networks by human and machine-based agents. This article includes a description of DCSS's underlying model and implementation, examples of its operation, and an evaluation of its strengths, weaknesses, and potential for future growth.

Key words: cooperative, design, conflict, concurrent, engineering, rationale

1. The challenge: supporting cooperation in design groups

Design has increasingly become a cooperative endeavor carried out by multiple agents with diverse kinds of expertise. The design of a car, for example, requires experts on potential markets, ease of manufacturability, safety regulations, available means for shipping vehicles, and so on. Many companies are currently studying or implementing approaches to facilitate cooperative design using so-called "concurrent engineering" or "simultaneous engineering" techniques (Perry 1990).

Conflict resolution is a critical component of the cooperative design process. Different agents, representing different portions (e.g., different product systems) or life cycle stages (e.g., design, manufacture, maintenance, and so on) of the design often have different notions concerning what kind of design is best. In one

domain studied, roughly 50 percent of the statements made by a group of collaborating designers involved either the identification or resolution of conflicts (Klein and Lu 1990). Ineffective management of conflicts can result in greatly increased product costs and reduced quality. In complex artifacts consisting of many interacting systems and/or in artifacts with many different life cycle stages, however, detecting and resolving conflicts effectively can be very difficult.

The development of tools and underlying theories for supporting conflict management in cooperative design has lagged, however, behind the growing needs for such work. While other aspects of cooperative activity have been studied in some depth (Malone, Fikes, and Howard 1983; Thorndyke, McArthur, and Cammarata 1981; Smith and Davis 1979; Corkill and Lesser 1983), conflict management has been only lightly explored. Work to date has significant limitations; most notably, these systems do not support task-level interaction and embody little or no conflict detection and resolution expertise.

The work described here represents the fruition of a three-stage program of research. The first stage involved studying how human designers cooperate to design artifacts (Klein and Lu 1990). The main lesson of this work was that one can identify general conflict resolution expertise applicable to a wide variety of design conflicts. The second stage involved creating a computational model and associated cooperative design "shell" that embodies these insights. This shell was instantiated into a system for cooperative Local Area Network (LAN) design consisting of seven machine-based design agents, each with differing areas of design expertise (Klein and Lu 1990). The third stage involved extending the cooperative design shell to allow human designers also to participate in the design process. To do so, a model for how human designers and their assistants can effectively interact had to be developed. The extended shell, called DCSS (the Design Collaboration Support System), has been used successfully to support cooperative LAN design with human and machine-based design agents. This work has thus come full circle: insights into human conflict resolution have been embodied into a computer system used by human designers.

The remainder of this article is organized as follows. Existing work on computer support for group conflict resolution is reviewed, examining both its contributions and limitations. We then consider the theory underlying DCSS and look at some examples of its operation. The article concludes with an evaluation of DCSS and directions for future work.

2. Contributions and limitations of existing work

Work relevant to support of group conflict management comes from artificial intelligence (AI) and related fields as well as the social sciences. A large body of work is devoted to analyzing human conflict resolution behavior (Coombs and Arrunin 1988). This work highlights the importance of conflict in group interactions but provides few prescriptions for how conflict resolution can be facilitated.

Moreover, much of this work focuses on issues specific to the psychology of human participants, rather than on the general nature of conflict resolution.

There is in addition work on supporting human interaction, including research on group consensus building and group decision support systems (GDSSs) (Sarin 1985; Nunamaker, Applegate, and Konsyaski 1987). Such work provides some structure for interactions among group members, but the conflict management expertise is expected to reside in the human participants. This is an inevitable consequence of the level of interaction of the users with the collaboration support system: since the system has at best a very shallow understanding of the contents of the participant's interactions, it can offer relatively little meaningful support.

To find work on systems that directly support conflict resolution, we need to turn to AI and related fields such as planning and design (Trice and Davis 1989; Sussman and Steele 1980; Fox and Smith 1984; Descotte and Latombe 1985; Brown 1985; Marcus, Stout, and McDermott 1987; Hewitt 1986; Wilensky 1983; Lander and Lesser 1988). Such work uses expertise encoded in the system itself to detect, understand, and resolve conflicts among design participants. While a broad range of research has taken place in this area, existing systems are limited in the extent to which they give conflict resolution expertise "first-class" status, i.e., represent and reason with it explicitly using formalisms as robust as those used for other kinds of expertise. In addition, this work has been applied only to machine-based design agents, so issues concerning the interface with human participants have been left unaddressed. In the following section our own approach to supporting conflict resolution among design agents, which we believe avoids the limitations of previous work in this area, is described.

3. DCSS: the design collaboration support system

The goal of DCSS is to help human and computational participants of cooperative design teams effectively detect, understand, and resolve conflicts among them. DCSS has a distributed architecture consisting of a collection of design agent/assistant pairs (Figure 1). Agents work by refining and critiquing a shared representation of the design, and can be either human or machine-based. Assistants are computer programs that help detect and resolve negative interactions between the actions of the different agents. In the current implementation, human designers work in front of dedicated workstations executing their assistant, while machine-based design agents are independent processes with incorporated assistant code. Agents can be based on different machines, work asynchronously, and communicate with each other over a network. The code currently runs on networked Symbolics Lisp Machines and is written in Common Lisp.

Assistants provides three kinds of services to their associated agents: *conflict detection* (helping design agents detect conflicts among each other); *conflict explanation* (suggesting one or more possible causes for a conflict); and *conflict resolution* (suggesting potential resolutions to conflicts once detected). In the sec-

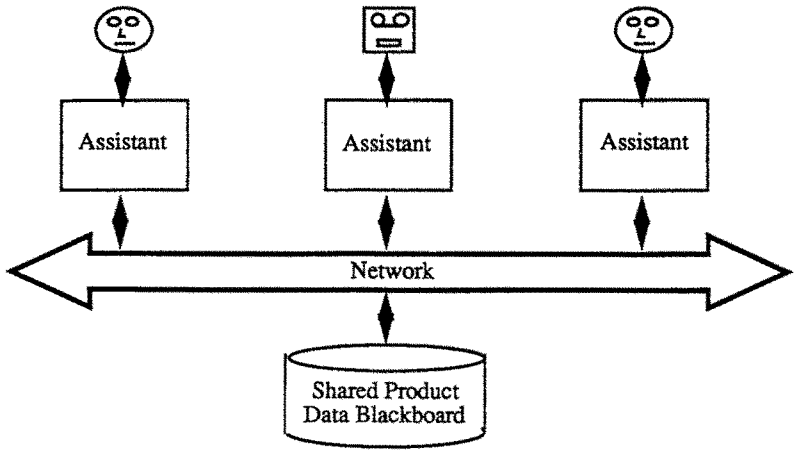

Figure 1. The DCSS architecture.

tions below, ways in which assistants provide each of these services are discussed in turn.

3.1. Detecting conflicts

The first service that DCSS assistants provide design agents is support in detecting conflicts among them as early as possible. DCSS supports this in two ways. First, it allows design agents to describe their design actions in terms of a least-commitment design model (Stefik 1981) that helps avoid unnecessary conflicts and allows early conflict detection. Second, it provides a range of tools for detecting conflicts once they occur. In the following sections we examine both.

3.1.1. Describing least-commitment designs. Describing design actions requires an effective interface between agents and their assistants, which implies that the interfaces should be based on a *task* rather than *implementation*-level model of the problem domain (Gruber 1989). Implementation level interfaces, used in many knowledge-based systems, allow the user to interact with the system only in terms of the entities (e.g., rules, facts, assumption-based truth maintenance system (ATMS) contexts) in which the system is implemented. Task level interfaces, by contrast, allow the user to interact with the system in terms of entities (e.g., resources, components, diagnoses) appropriate to the task being performed. If design agents express their actions using a task-level model, then to the extent that the model is accurate, it allows these agents to communicate in a domain-natural but computer-interpretable way. In the paragraphs below we briefly describe DCSS's task-level model of cooperative design as well as the interface to human design agents based on this model. For a more complete description see Klein (1992).

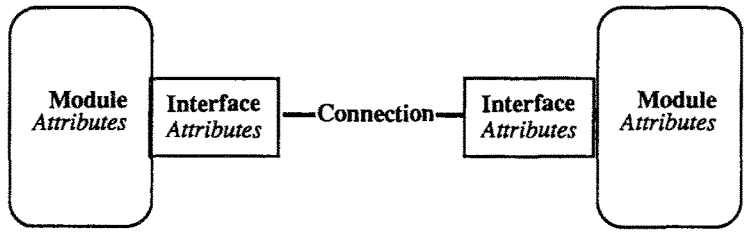

Figure 2. The design description scheme.

In DCSS, physical artifacts are viewed as collections of modules, which can represent entire systems, subsystems, or their components, each with characteristic attributes, whose interfaces (with their own attributes) are connected by connections of a given type. The resources used by a module (e.g., cost, weight) are represented using a special class of attribute (Figure 2).

A computer, for example, can be described as a set of very large scale integration (VLSI) chip modules with attributes describing their functionality, power consumption, and so on. The modules' interfaces (pins) are connected to each other via (electrical) connections realized as deposited wires. At another level, we can view an entire board as a module connected via a connector to the bus and indirectly to other boards. At yet a higher level, a computer can be viewed as a set of interacting subsystems including disk I/O, memory and the central processing unit (CPU). The interfaces and connections at this level describe the data and control protocols connecting these systems. Hydraulic systems, similarly, can be viewed as pipe, switch, tank, and pump modules with hydraulic connections.

Artifact descriptions are refined using an least-commitment iterative refinement process. An artifact description starts as one or more abstract modules representing the desired artifact (e.g., "airplane," "computer," or "software application") with specifications represented as desired values on module attributes (e.g., "passenger capacity should be > 350"). This is refined into a more detailed description by constraining the value of module attributes, connecting module interfaces (to represent module interactions), decomposing modules into submodules, and specializing modules by refining their class (Figure 3). Specifications and attribute values are described using a constraint language (Stefik 1981; Klein 1991).

If we were designing an airplane, for example, we might decompose the top-level "airplane" module into wing, tail, and body section modules as well as electrical, hydraulic, and mechanical subsystem modules. Interactions between modules (e.g., physically connected components) are represented as connections between module interfaces.

The least-commitment nature of the DCSS design model has two important advantages. First of all, unnecessary conflicts between different agents are avoided, since the designer is not forced to choose arbitrarily from a set of acceptable alternatives simply to make a definite commitment. In addition, conflicts can po-

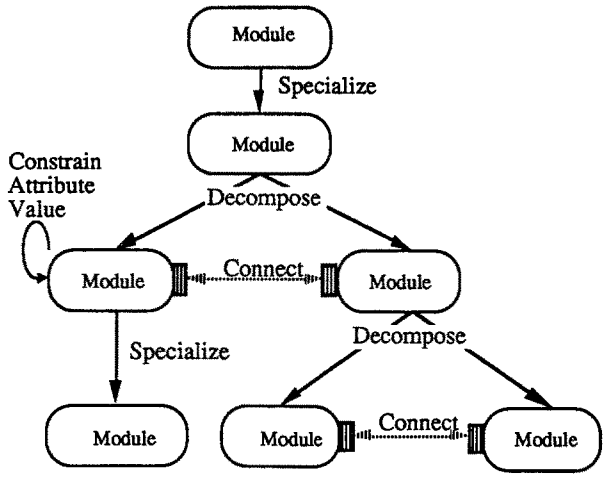

Figure 3. The design refinement process.

tentially be detected while the design is still indefinitely described, minimizing the time before a conflict is detected and thereby minimizing the amount of design effort expended on unviable design alternatives. Most CAD (computer-aided design) systems currently available, however, do not support least-commitment design descriptions. DCSS can still help detect, explain, and resolve conflicts with such systems, but at the cost of increased rework in the design process.

This design model is based on classical systems engineering work as well as AI models of artifact design (Tong 1987; Klein 1991) and planning (Stefik 1981). These models have been applied successfully to electrical, electronic, hydraulic, and mechanical systems as well as software.

All human design agents are provided with a direct-manipulation WIMP (window icon menu pointer) graphical interface like that familiar to users of Macintosh and Windows systems (Figure 4). Using this interface, the user can create windows that present some desired subset of the design data from one of many possible perspectives, each designed to highlight different aspects of the design description. In Figure 4, for example, the user is viewing the versions graph (lower right), the artifact design in one of the versions (upper right), a description of one component in that design (lower left), and a description of the plan used to produce that component (upper left). These windows dynamically update themselves whenever the subset of the design data they view changes, so they are continuously up to date.

The printed representation of every assertion is mouse-sensitive; a menu of options that makes sense in the current context for that assertion appears when it is clicked on using the pointing device. There are two classes of options for any assertion: perspective creation and editing. Perspective creation options create windows that view the assertion from a given perspective. When clicking on the

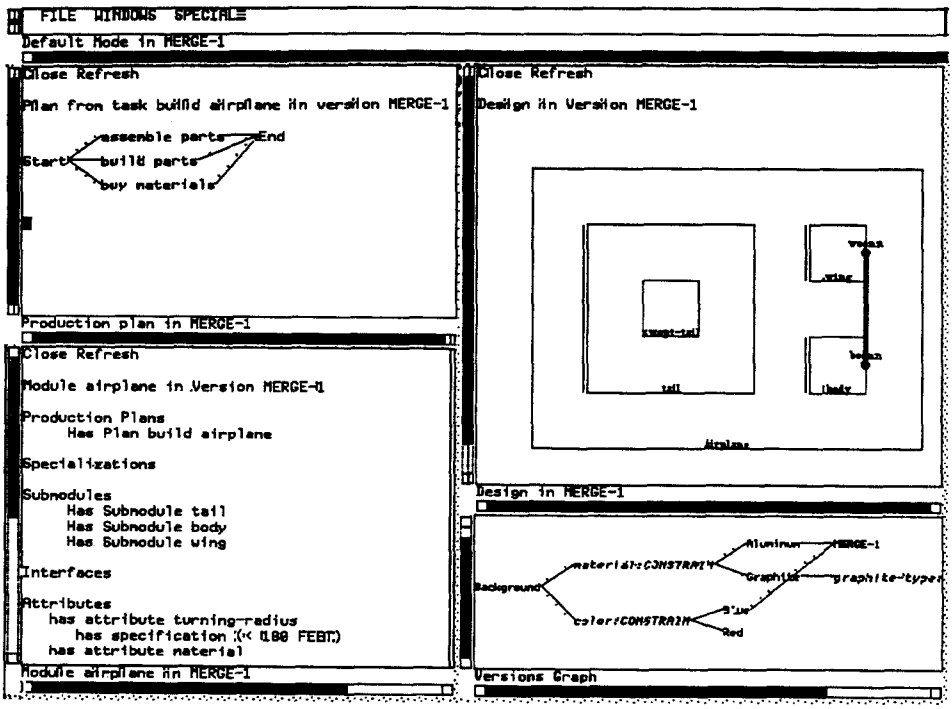

Figure 4. Example of the DCSS human agent interface in use.

top-level task of a plan, for example, one can create windows that view it either as temporally ordered leaf tasks or as a task decomposition hierarchy. Editing options allow one to update the design description; for any assertion the menu will include a list of all the types of statements one can make about that assertion. To add an attribute to a module, for example, one clicks on the module and selects the "Define Attribute" option; the user is then prompted for an attribute name. To create a design requirement one clicks on a module attribute, selects the "Define Requirement" option and enters a constraint representing the desired attribute value. To connect two module interfaces, one selects the "Make Connection" option for one interface and then selects the interface to which to connect.

3.1.2. Conflict detection tools. In general, when different agents give incompatible specifications for a given design component, or one agent has a negative critique of specifications asserted by another agent, we can say that a conflict has occurred. Support for conflict detection is provided in two ways (Figure 5).

Some conflicts can be detected in a domain-independent fashion: for example, incompatible constraints on a given component feature immediately imply a conflict. Domain-specific conflict detection is also supported. Design experts are, in general, very knowledgeable about situations that represent problems with a de-

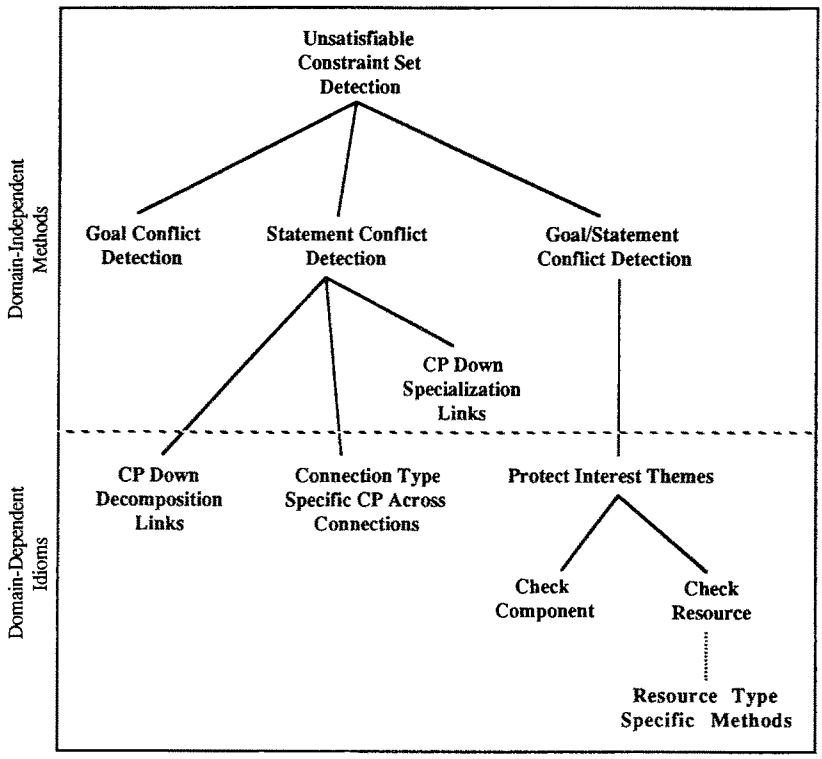

Figure 5. Conflict detection tools provided by DCSS.

sign from their particular perspectives. We have found that the techniques used by human designers to detect such conflicts are often instantiations of a relatively small set of stereotyped conflict detection (CD) methods we call CD "idioms" (Klein 1991). DCSS allows designers to quickly instantiate CD idioms with situation-specific values, thereby creating "demons" that search for a particular kind of design problem. Whatever technique is used, the assistant automatically updates the design rationale so that the design decisions underlying a conflict can be readily determined.

Domain-independent methods In a least-commitment model such as that used in DCSS, all conflicts eventually manifest as unsatisfiable constraint sets on a given feature. Underlying all CD methods, therefore, is an unsatisfiable constraint set detection mechanism. This is implemented as constraint combination (i.e., simplifying constraint sets by finding their consequences) coupled with detection of individual unsatisfiable constraints. For example, the range constraints (from 10 to 12 and from 14 to 18) intersect to produce an unsatisfiable constraint. Similar techniques are available for other constraint types.

Built on top of this is support for detecting conflicts between the two kinds of

assertions made by design agents: constraints on component features ("statements") and goals. There are three kinds of conflicts at this level: conflicts between achieve-value goals (inconsistent desired constraints on a component feature), conflicts between statements (inconsistent constraints on a component feature), and conflicts between achieve-value goals and statements (inconsistent desired and actual constraints on a component feature).

Constraints asserted by design agents can propagate along the links created during the design process and cause conflict over design features involving some other portion of the design. To find the complete set of constraints implied by a given design description on a component feature, one unions the constraints asserted directly on the component feature with those that propagated over links with related design features. In the DCSS design model, configuration and specialization links propagate constraints domain-independently:

- *Configuration links.* Configuration links interrelate component features within or between components. For example, the constraint on the voltage drop across a resistor is related by a configuration link to the constraint on the current through the resistor. This kind of link is used widely in constraint network systems (e.g., Sussman and Steele 1980; Marcus, Stout, and McDermott 1987).
- *Specialization links.* Specialization links connect components and their specializations. All constraints that apply to an abstract component apply to its specialization, so all constraints are propagated down. For example, all the constraints that apply to an abstract "Memory Module" apply to the particular memory technology chosen (e.g., magnetic media hard disk) to implement it.

Domain-dependent idioms Built on top of the domain-independent CD methods are the CD idioms. When a design agent wants to describe a piece of CD expertise, he/she often can simply use a pre-existing idiom, filling in the domain-specific details needed to instantiate the idiom into an active entity. CD idioms found useful to date include:

- *Propagation across connection links.* Connection links connect the interfaces of modules to each other. The constraints that propagate through a connection depend on the kind of connection. An electrical connection, for example, propagates constraints on voltage and current. A physical connection between rotating axles propagates constraints on rotation speed and torque. Since DCSS includes the notion of typed connections, it can use type information to determine what constraints to propagate. DCSS's current connection type hierarchy is given in Figure 6.

An agent merely need specify the type of a connection by selection from the above hierarchy; the assistant uses this information from then on to propagate constraints appropriately.

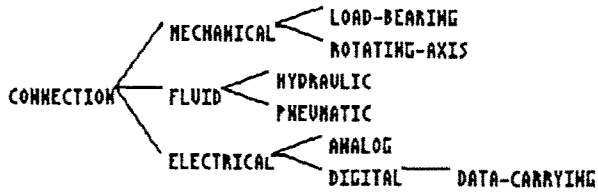

Figure 6. DCSS connection types hierarchy.

• *Propagation down decomposition links.* Decomposition links connect abstract modules to the several modules that realize them. While the propagation of constraints down decomposition links is domain-dependent, one can often use the mapping between abstract and more specific connections to simplify describing such propagation. For example, an abstract LAN trunk module is usually connected to abstract LAN trunk connector modules. When the LAN Trunk module is refined to a particular trunk technology, the constraints on the connections to the abstract module's interfaces apply to the corresponding connections in the specific trunk instance. Domain-specific knowledge of how connections are mapped over a decomposition step is called "structural" knowledge in MICON (Birmingham and Siewiorek 1989).

• *Protect interests themes.* Protect interests themes monitor the current design state for a state threatening to the interests of the design agent, and make assertions that lead to a conflict being asserted if the threatening design state should ever come to pass. We have identified two useful protect-interest CD idioms to date: *check-module* and *check-resource*.

Check-module idioms check that interests relating to individual modules are satisfied, and have the following format:

IF there is an instance of <module>
THEN create the goal to achieve <desired value> for <feature> of <module>
create the goal to find the actual value for <feature> of <module>

To instantiate such an idiom, one simply selects "Create Check Module Theme" from the options for the module feature and enters the value desired for that feature.

Check-resource idioms look for situations where some kind of resource budget is exceeded. Resources in most design situations are limited in some way, and it is useful to create budgets representing limits on how much of a given kind of resource should be utilized for a given portion of the design. This idiom works by creating a goal to summarize the total usage of a given resource, as well as the constraint that the actual utilization should not exceed the budget:

IF there is a <resource> provided by <module>
THEN assert the constraint that the amount of <resource> committed by <module> is less than or equal to the amount of <resource> provided
create a goal to find the amount of <resource> committed by <module>

To instantiate such an idiom, a design agent need merely specify a budget and select the "Create Check Resource Theme" option for the resource of interest. From that point on, if resource utilization exceeds the budget, a conflict will occur.

Resource overutilization, however, is found differently depending on the kind of resource involved. For example, overutilization of a monetary budget can be found simply by summing individual expenditures and comparing them with the budget. Detecting space overutilization is somewhat more complicated; even in the two-dimensional case (i.e., checking for adequate floor space), equipment whose total area does not exceed the available floor space may not fit in a given area due to the particular shapes of the equipment involved. A functional resource that is not used up over time but can only support a finite number of simultaneous users (e.g., a computer terminal) has resource overutilization detected in yet a third way. DCSS supports this by typing resources (Figure 7) and providing conflict detection idioms and resource usage analyzers for each type.

Note that conflict detection idioms can be heuristic or "shallow" in nature. This can be useful if more exhaustive analysis is computationally expensive but raises the possibility that the conflict may be spurious, resulting from the limitations of the analysis procedure.

The models described above represent a superset of the conflict-detection methods utilized in the other approaches to conflict management, and provide a complete set of basic mechanisms for conflict detection. The notion of an abstract resource-type hierarchy has been previously discussed (Wilensky 1983) as has the conflict-detection advantages of using a least-commitment design model (Stefik 1981). Previous work has not, however, discussed the use of domain-independent connection-type and resource hierarchies to support conflict detection, or the notion of instantiable CD idioms.

3.2. Explaining conflicts

The second important service DCSS assistants provide design agents is support for understanding conflicts once they have been detected. Conflict causes can, in general, be difficult to determine given their manifestations. When the manifestation of a conflict is an exceeded weight budget on an airplane system, for example, the cause may be the choice of an inappropriate material for the structural

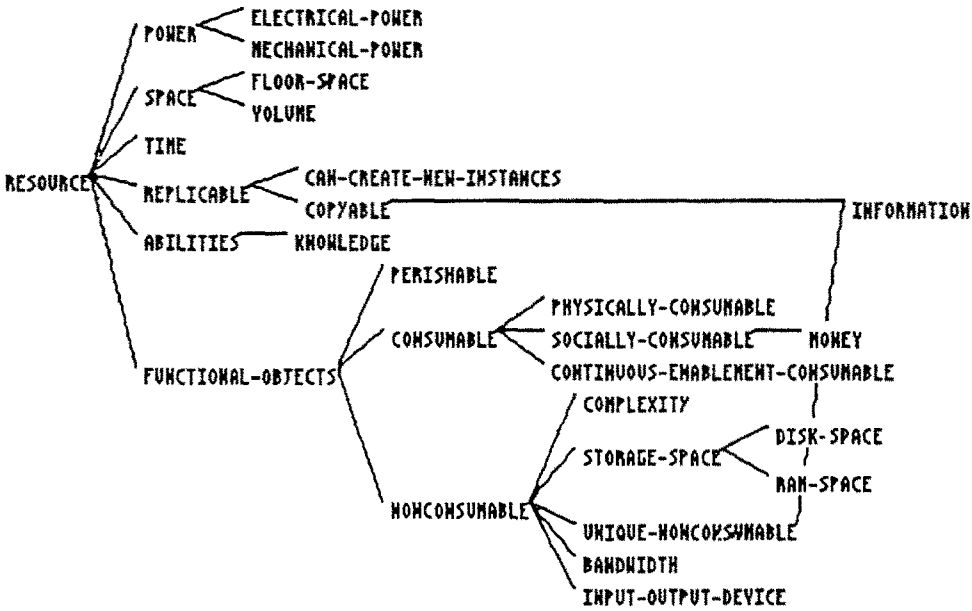

Figure 7. DCSS resource types hierarchy.

members, failure to consolidate redundant components, and so on. Our studies of conflict management in human design teams (Klein 1990) has revealed, however, that there are only a limited number of fundamentally different conflict causes. Conflict causes can, in fact, be arranged into a *taxonomy* with the most abstract and inclusive classes at the top, and with increasingly specific causes further below (Figure 8).

This taxonomy falls into three layers. The top layer includes conflict causes that can appear in any imaginable design domain (e.g., "picked wrong plan to achieve goal"). The second layer includes causes that, while not completely generic, apply to broad classes of design domains. The conflict cause "unwanted resource propagation through conduit," for example, applies to any design domain where resources of some kind (e.g., oil, electricity, information) propagate through conduits of some kind (e.g., pipes, wires, networks). The third layer includes classes specific to particular domains (e.g., "excessively tight tolerance leads to unnecessary manufacturing tool"). DCSS assistants presently include a taxonomy of 115 different conflict causes.

DCSS assistants help their design agents understand the cause of a conflict by finding the conflict classes that match the conflict under consideration, i.e., by heuristic classification (Clancey 1984). Every conflict class has a set of defining characteristics expressed as questions and acceptable answers using an abstract "query language." When a conflict occurs, the DCSS assistant that detected the conflict traverses the conflict class taxonomy, asking the design agents involved

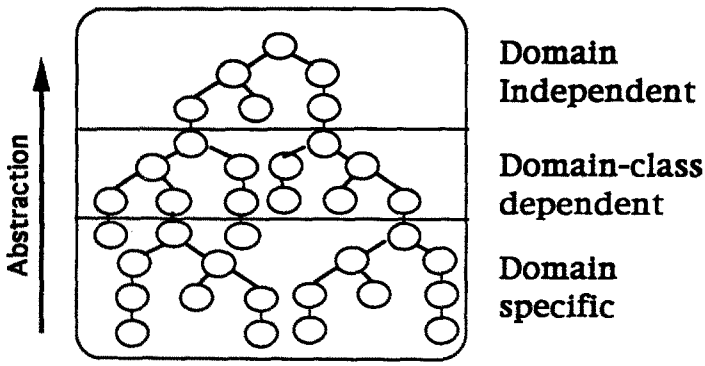

Figure 8. A conflict-cause taxonomy.

the questions associated with each class and tagging those classes whose defining characteristics were satisfied by the answers returned. Since preconditions often only approximate the true conditions for a class being matched, these explanations are hypotheses rather than provably correct.

Consider the following example. Imagine we have a conflict arising during the design of a local area network (LAN) that manifests as unsatisfiable constraints on the choice of the LAN physical topology. One of the conflict classes considered by the assistant has defining conditions we can paraphrase as follows: "A design agent is unable to find a module in a module database that satisfies some set of constraints because the constraints were too rigid."

The dialogue that results from testing for a match is the following:

Assistant	Design Agent
What is the logical support for *conflict-1*?	Assertion-1: The physical-topology of LAN-trunk-1 is nil
Physical-topology is of type module?	Yes
What is the logical support for *assertion-1*?	Strategy-1
Strategy-1 is of type database-search?	Yes
What is the logical support for *Strategy-1*?	Assertion-2: The media of LAN-trunk-1 is thin-baseband-coax
	Assertion-3: The protocol of LAN-trunk-1 is token-ring
Is *Assertion-2* relaxable?	Yes
Is *Assertion-3* relaxable?	Yes

Note that while the questions representing conflict class characteristics are always expressed in abstract domain-independent terms, later questions can have

parts of them filled in (here represented in italics) based on the agent responses to previous questions.

Since the defining conditions for this conflict class were all satisfied, the assistant will offer as a potential explanation the following: "Design agent *agent-1* can't find a *physical-topology* for *LAN-trunk-1* because the constraints on *media* of *LAN-trunk-1* and *protocol* of *LAN-trunk-1* are too rigid."

Since there are often several potential causes for a given conflict, the DCSS assistants will offer a list of such explanations for every conflict. For a more complete description of the DCSS conflict explanation process, see Klein (1991).

Typically, many questions (on the order of 100 or so, depending on the conflict) need to be answered per conflict explanation process. This clearly would place an excessive burden on the human design agents if they had to answer them all. The strategy taken in DCSS is to make assistants answer as many as possible of these questions themselves. Using this scheme, generally only two or three questions are passed on to human design agents per explanation.

The questions asked by assistants during the explanation process fall into four classes: value retrieval, type matching, rationale, and option-exploring. Value retrieval questions simply ask for design entity attribute values, and can be answered by examination of the design description. Type matching questions ask if a given design entity is of the abstract type mentioned in a conflict class condition. This can be supported by providing assistants with domain-specific ISA hierarchies. Option-exploring questions ask about whether there are other options for a given design decision; since they require considerable domain expertise to answer, they are passed on to the design agents. Rationale questions ask about the logical support for design decisions.

Design rationale is captured using a rationale language based on a task-level model of how designers reason. Design rationale consists, essentially, of the logical relationships between different design decisions. The rationale for a product geometry decision, for example, consists of the requirements it attempts to satisfy and the other geometry decisions on which it logically depends. A manufacturing plan decision, similarly, is justified in terms of supporting decisions like project schedule and geometry decisions. Since there are different kinds of decisions (e.g., requirements, design descriptions, control choices) as well as different ways they can be related, this implies the need for a language of typed relationships, or *links*.

DCSS assistants allow design agents to describe their design rationale using a typed link language built from existing work in this area (Lee and Lai 1991; Yakemovic and Conklin 1990; McCall 1987; MacLean et al. 1991). In this language, design decision interdependencies are represented as *claims*. Any claim can serve as part of the rationale for another claim, so we can make claims about the design, claims describing the rationale for design decisions, claims describing why we should believe this rationale (or not), and so on. The DCSS rationale language is summarized in Figure 9 below: the plain font items represent types of design decisions, while the italic labeled arrows represent allowable kinds of typed links between them, pointing from the sources to the targets.

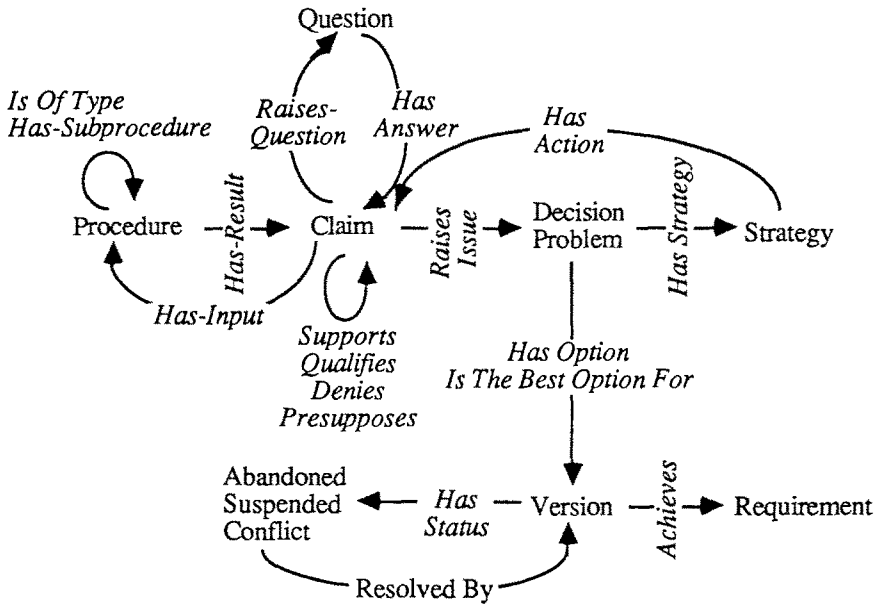

Figure 9. DCSS design rationale language.

A human design agent can build up a rationale description using a simple extension of the user interface provided for describing the designs themselves. The menus brought up by selecting an assertion with the pointing device include context-sensitive lists of the types of links allowable starting from that assertion. To create a link, one simply selects the link type and the assertion that is the target of the link. The net result of describing design rationale in this way is a graph of decisions interrelated by typed rationale links. Consider the following rationale for the choice of standard manufacturing machinery to produce an airplane wing (Figure 10).

Standard machinery was chosen because the wing material is aluminum. This choice of material has two separate lines of support. The designer's strategy of using inexpensive materials to meet a cost requirement provides one justification. The constraints on wing loading and weight also suggest aluminum, based on reference to an engineering table.

While design agents can, of course, choose to describe the rationale for their design decisions only when asked at conflict explanation time, design agents are asked to describe as much as possible the rationale for their design decisions *as they make them*. This is done for two reasons. Reconstructing the rationale for a design decision after it has been made can sometimes be quite difficult. More importantly, expressing design rationale is useful for many tasks other than conflict resolution, such as explanation, learning, and redesign (Kellogg, Mark, and McGuire 1989; Acosta, Huhns, and Ligh 1986; Brown 1985; Simoudis 1988; Mos-

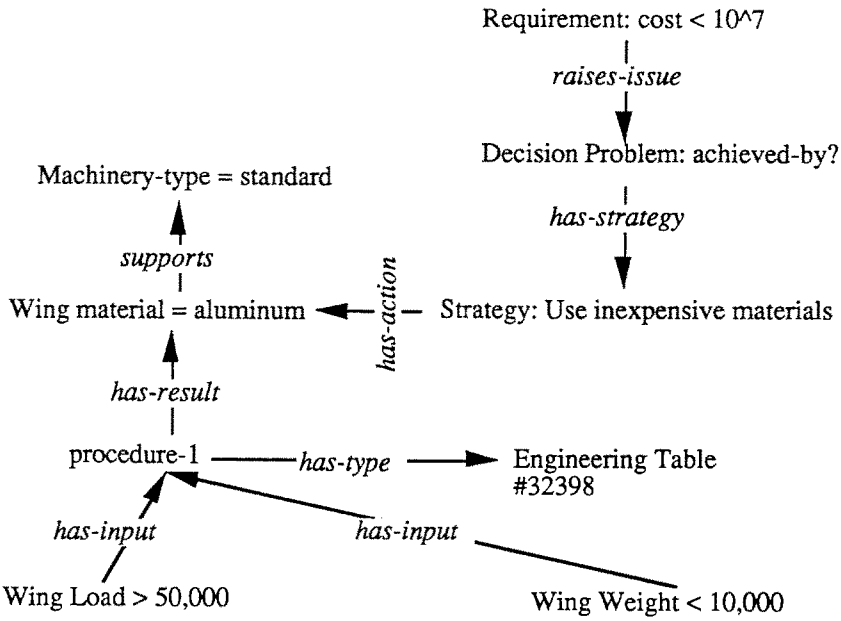

Figure 10. An example of design rationale.

tow and Barlay 1987; Howard et al. 1986; Batali 1983). For a complete description of the DCSS rationale language and user interface, see Klein (1992).

3.3. Resolving conflicts

The third of the services provided by DCSS design assistants is generating suggestions for resolving conflicts once detected and explained. Just as there are a limited number of fundamentally different kinds of conflict causes, there are a limited number of different strategies for resolving conflicts with a given cause. Consider the following examples:

FOR conflict cause: unwanted resource propagation through conduit
IF a filter for unwanted resource exists
THEN add filter to conduit

FOR conflict cause: picked wrong plan for goal
IF an untried alternate plan exists
THEN try an alternate plan

FOR conflict cause: unable to find a module that satisfies excessively rigid constraints

IF the constraints are relaxable such that a suitable module can be found
THEN relax the constraints accordingly

Conflict resolution strategies include a set of applicability preconditions expressed using the same query language used for conflict class preconditions, as well as the strategy itself expressed in an "action" language consisting of four primitive action types (modify plan output, modify plan procedure, add detail, and pick plan for goal). If the applicability preconditions for a strategy are met, the abstract strategy is instantiated using domain-specific information provided by the design agents. Consider the case where a conflict was diagnosed as being due to unwanted resource propagation (e.g., sunlight) through a conduit (e.g., a window). The general resolution strategy for this conflict ("add filter to conduit") is instantiated via the following dialogue:

Assistant	**Design Agent**
Is there such a thing as a filter for *sunlight* propagating through a *window*?	Yes
What *filters* are available for *sunlight* through *windows*?	Window shades, window overhangs.
Try one of the following resolutions: • Add the detail *window shade* to the *window* • Add the detail *overhang* to the *window*	OK—I'll try "add window shade."

Note that a given strategy can be instantiated into several different suggestions. The list of all suggestions generated for a conflict is ordered using domain-independent heuristics (e.g., "prefer the suggestion derived from the most specific explanation," "make minimum change to design," "avoid abandoning goals") and then presented to the design agents. The design agents can then select a strategy to execute; this strategy is then automatically added by the assistant as the rationale for the conflict resolution actions they subsequently take.

4. Lessons learned and future work

This article described a system called DCSS developed to meet the challenge of conflict management in design teams with both human and machine-based agents. Work on this system has produced two key lessons:

• *Conflict management isn't as complicated as it seems.* While the detection, understanding, and resolution of conflicts seems to present a bewildering array of

challenges, the relatively simple abstract underlying structure of this process makes it possible to provide computer-based support. In particular, computer-based assistants with generic cooperation expertise can work successfully with domain-knowledgeable machine or human design agents in many different domains.

• *Conflict management requires reflective and flexible design agents.* In order to participate in the DCSS conflict management process, the design agents have to be able to *describe* their actions using a least-commitment approach, *explain* their actions and options, and *modify* their actions when appropriate. This implies requirements for "cooperation-enabled" computer-based design agents as well as for interfaces to human design agents. A graphical user interface based on a "task-level" model of the design process seems well suited to meeting the latter challenge.

There are several potential directions for further development of DCSS. One set of issues concerns the basic system functionality. The assistants need to be enhanced so they can support explanation and resolution of multiple, related, simultaneous conflicts. The system should be validated in a number of differing design domains; this is likely to require additional acquisition of generic conflict causes and associated resolution strategies. The overhead of describing design decision rationale needs to be reduced relative to the benefits provided to the design agents; this is an area of active research in the rationale capture-research community.

In addition, we need to address the issue of making this technology available in actual cooperative design teams. For this to happen, advances need to be made on several fronts. Current product data and/or interapplication link standards need to be augmented to include a design rationale representation. Design application (e.g., computer-aided design (CAD) or computer-aided process planning (CAPP) system) user interfaces need to be updated to allow users to collaboratively view, edit, and link different kinds of shared product data. One approach is to enhance existing design tools so they can provide additional product data display formats (e.g., add displays for manufacturing data to a CAD tool) as well as allow linkage among this data. Another approach is to provide support at the operating system level, extending in effect the cut-and-paste metaphor used in the Macintosh operating system to support creation of cross-application rationale links. We are currently evaluating both approaches for viability in the context of Boeing's computing environment.

References

Acosta, R.D., M.N. Huhns, and S.L. Liuh. (1986). "Analogical Reasoning for Digital System Synthesis." In *Proceedings ICED*, IEEE, pp. 173–176.

Batali, J. (1983). "Dependency Maintenance in the Design Process." In *IEEE Int Conf Computer Design: VLSI in Computers*, pp. 459–462.

Birmingham, W.P., and D.P. Siewiorek. (1989). "Automated Knowledge Acquisition for a Computer Hardware Synthesis System." *Knowledge Acquisition* 1, 321–340.

Brown, D.C. (1985). "Capturing Mechanical Design Knowledge." *American Society of Mechanical Engineers CIME 1985.*

Clancey, W.J. (1984). "Classification Problem Solving." *AAAI-84,* pp. 49–55.

Coombs, C.H., and G.S. Avrunin. (1988). *The Structure of Conflict.* Lawrence Erlbaum Associates.

Corkill, D.D., and V.R. Lesser. (1983). "The Use of Meta-Level Control for Coordination in a Distributed Problem Solving Network." *IJCAI-83,* pp. 748–756.

Descotte, Y., and J.C. Latombe. (1985). "Making Compromises Among Antagonist Constraints in a Planner." *Artificial Intelligence* 27, 183–217.

Fox, M.S., and S.F. Smith. (July 1984). "ISIS—A Knowledge-Based System for Factory Scheduling." *Expert Systems.*

Gruber, T.R. (1989). "A Method for Acquiring Strategic Knowledge." *Knowledge Acquisition* 1(3), 255–277.

Hewitt, C. (1986). "Offices Are Open Systems." *ACM Transactions on Office Information Systems* 4(3), 271–287.

Howard, A.E., P.R. Cohen, J.R. Dixon, and M.K. Simmons. (1986). "Dominic: A Domain-Independent Program for Mechanical Engineering Design." *Artificial Intelligence* 1(1), 23–28.

Kellog, C., W. Mark, J.G. McGuire, et al. (1989). "The Acquisition, Verification and Explanation of Design Knowledge." *SIGART Newsletter* 108, 163–165.

Klein, M., and S.C.Y. Lu. (1990). "Conflict Resolution in Cooperative Design." *International Journal for Artificial Intelligence in Engineering* 4(4), 168–180.

Klein, M. (1991). "Supporting Conflict Resolution in Cooperative Design Systems." *IEEE Systems Man and Cybernetics* 21(6), 1379–1390.

Klein, M. (1992). "DRCS: An Integrated System for Capture of Designs and Their Rationale." In *Proceedings of Second International Conference on Artificial Intelligence in Design,* Carnegie Mellon University, Pittsburgh, PA.

Lander, S., and V.R. Lesser. (1988). "Negotiation To Resolve Conflicts Among Design Experts." Tech. Report Dept. of Computer and Information Science, University of Massachusetts at Amherst.

Lee, J., and K.Y. Lai. (1991). "What's In Design Rationale?" *Human-Computer Interaction* 6 (3–4).

MacLean, A., R. Young, V. Bellotti, and T. Moran. (1991). "Questions, Options and Criteria: Elements of a Design Rationale for User Interfaces." *Journal of Human Computer Interaction: Special Issue on Design Rationale,* 251–280.

Malone, T.W., R.E. Fikes, and M.T. Howard. (1983). "Enterprise: A Market-Like Task Scheduler for Distributed Computing Environments." Tech. Report Cognitive and Instructional Sciences Group, Xerox Palo Alto Research Center, Palo Alto, CA.

Marcus, S., J. Stout, and J. McDermott. (1987). "VT: An Expert Elevator Designer." *Artificial Intelligence Magazine* 8(4), 39–58.

McCall, R. (1987). "PHIBIS: Procedurally Heirarchical Issue-Based Information Systems." In *Proceedings of Conference on Planning and Design in Architecture,* Boston, MA: ASME.

Mostow, J., and M. Barley. (1987). "Automated Reuse of Design Plans." In *Proceedings ICED,* IEEE, 632–647.

Nunamaker, J.F., A. Applegate, and K. Konsynski. (1987). "Facilitating Group Creativity: Experience with a Group Decision Support System." In *Proceedings of the Twentieth Hawaii International Conference on System Sciences,* pp. 422–430.

Perry, T.S. (1990). "Slashing Development Time: Combining Technology with Teamwork." *IEEE Spectrum* 27(10), 61–78.

Sarin, G. (1985). "Computer-Based Real-Time Conferencing Systems." *IEEE Computer,* October.

Simoudis, E. (1988). "Learning Redesign Knowledge." In *Banff Knowledge Acquisition for Knowledge-Based Systems Workshop,* Banff, British Columbia, Canada: AAAI.

Smith, R.G., and R. Davis. (1979). "Cooperation in Distributed Problem Solving." *IEEE Proceedings of the International Conference on Cybernetics and Society,* pp. 366–371.

Stefik, M.J. (1981). "Planning with Constraints (Molgen: Part 1 & 2)." *Artificial Intelligence* 16(2), 111–170.

Sussman, G.J., and G.L. Steele. (1980). "Constraints—A Language for Expressing Almost-Hierarchical Descriptions." *Artificial Intelligence* 14, 1–40.

Thorndyke, P., D. Mcarthur, and S. Cammarata. (1981). "Autopilot: A Distributed Planner for Air Fleet Control." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence,* pp. 171–177.

Tong, C. (1987). "AI in Engineering Design." *Artificial Intelligence in Engineering* 2(3), 130–166.

Trice, A., and R. Davis. (1989). "Consensus Knowledge Acquisition." Tech. Report Information Technologies Group, MIT School of Management, Cambridge, MA.

Wilensky, R. (1983). *Planning and Understanding.* Addison-Wesley.

Yakemovic, K.C.B., and E.J. Conklin. (1990). "Report on a Development Project Use of an Issue-Based Information System." In *CSCW 90 Proceedings,* pp. 105–118.