faced by an AT&T organization that produces and sets of telecommunications guidelines, and managing any development work resembling the components and reusable artifacts into work at a high level, specifying information relevant features, reusing existing developed and implemented a prototype knowledge-based system that supports users in browsing and searching for related artifacts primarily from existing components. This type of task requires locating components and reusing existing artifacts, as well as assembling a new artifact that satisfies domain constraints and ensures that the delivery construction environment the graphical construction environment, the CD-CON. CD-CON is a human-computer collaboration with developers in cooperation with an AT&T product organization. They created a production version of the tool by adding and deleting graphical objects and changing the manipulation of objects in a shared graphical workspace.

Over the last 10 years, rapid growth in the number of calls that will be processed, minimum reliability of the switch, and standards that must be met. Customer response task involves assembling a system that organizes collaboration around features to satisfy customer requirements; the set of features running on a switching system constitutes the program that controls how the switch behaves. It is important to note that the customer response task is a "running system" for an application that requires new development work. Over the last 10 years, rapid growth in the number and type of features. In many of the customer requirements have led to a corresponding increase in the number and type of features. Inadequate representation of the ISCBU responds by creating a requirements document to AT&T (and its competitors). Requirements specify desired functionality and constraints such as number of calls that will be processed, minimum reliability of the switch, and standards that must be met. A feature is an abstraction of software functionality that specifies a coherent piece of behavior; the set of features running on a switching system is the 5ESS(TM). The ISCBU's customers are international telephone operating companies. When a new customer joins, the limits of the systems and processes for managing feature information were reached, leading to a need for a more robust system. The ISCBU is an AT&T organization that develops and markets a very large, complex switching system. The ISCBU serves a large number of customers with a diverse range of needs. It is important to note that the capabilities of the switch, and standards that must be met. The complete prototype served as a testbed for experimentally developing, testing, and revising our research ideas. Requirements specify desired functionality and constraints such as number of calls that will be processed, minimum reliability of the switch, and standards that must be met. The complete prototype served as a testbed for experimentally developing, testing, and revising our research ideas.
These problems make it very difficult to respond to customers quickly and accurately. The ISCBU initiated the RDDI (Rapid Definition and Deployment Initiative) project in response to these problems. By analyzing the existing feature base, RDDI achieved a consistent, customer-oriented definition of features and associated information.

A feature is the fundamental unit of customer functionality. Each feature has an associated set of text attributes, such as description, customer benefit, and standard compiled with, that are useful in searching for features that meet customer requirements. An option is a choice that determines how a feature operates. A setting is a legal value for an option; each option has a default setting. A bound feature is a feature with settings for all its options that has been tested and delivered. There is a close analogy between these concepts and object-oriented terminology: a feature corresponds to a class, options to the slots of a class, settings to the range of possible fillers for a slot, and a bound feature to an instance of a class. One important point is that it is bound features (rather than features) that are delivered to customers—the options of a feature specify a space of possible functionality, and setting the options (resulting in a bound feature) specifies a point in that space.

There also are various types of dependencies between these entities. Dependencies are co-occurrence constraints on the inclusion of features in a Customer Delivery, the set of features to be delivered to a single customer. There are two main types of dependencies, required-by and inconsistent-with. For example, setting a given option to a particular value may require setting a second option to a certain value (from its range of settings), and including a certain feature in a Customer Delivery may require including another feature or, conversely, may rule out including another feature. Dependencies may relate any combination of features, options, and option settings.

An application is a particular way the SESS may be used. Each Customer Delivery must specify one or more applications. Each application has two types of dependencies on features: basic features are required for the application, and premium features offer enhanced functionality. Thus, selecting an application for a Customer Delivery automatically determines a large set of features.

The goal of our partnership with RDDI was to develop a system to support the customer response task. Through much discussion, brainstorming, and scenario generation we defined key requirements:

- **Information location** – The set of features and associated entities form a large, complicated information space. It must be easy for users to locate objects in the space that satisfy their requirements. However, ISCBU information access tools—several Unix commands with complex options that displayed as textual tables—were inadequate.

- **Reuse** – The advantages of reuse are well-known, including saving work, adhering to standards, and promulgating good design practice. In ISCBU customer response, feature lists created by other individuals working on the current project, feature lists delivered to customers in the past, and standard packages of features all are candidates for reuse. However, there was no support for reuse in the ISCBU. The fundamental problem was that feature lists were simply text documents, stored in various locations in the file system. Formalizing feature lists and representing them in a central repository are a prerequisite for reuse.

- **Consistency maintenance** – Once dependencies between components (here, features) have been formalized, the system should assume responsibility for maintaining them, ensuring that artifacts (here, Customer Deliveries) are complete and consistent. However, maintaining the consistency of a Customer Delivery was a manual process, tedious and error prone. Errors that persist into development, testing, or (worst of all) after delivery of the switch are very costly.

- **High level, partial specification** – Users should be able to specify what they know at a high level and as they know it, with the system filling in details and keeping track of any partial specifications that eventually must be completed. However, users had to specify every detail of a Customer Delivery manually.

These requirements are not unique to this task, but are intrinsic to software construction tasks in general.

## 3 The Customer Delivery Construction Environment

The Customer Delivery Construction Environment, CD-CON is a knowledge-based graphical tool that supports the software construction task. In its current version, it works with a knowledge base of feature information. Users browse and search the knowledge base to locate features or existing feature lists to add to the Customer Delivery they are constructing. As users add information to and delete information from the delivery, the system applies feature dependency rules to maintain consistency. Figure 1 shows CD-CON and is referred to as we discuss system behavior.

### 3.1 Knowledge representation and reasoning

CD-CON uses Classic[3, 4] to represent feature information. Classic is a description logic that allows classes and individual objects to be described in terms of necessary and sufficient conditions. Classic can automatically determine subsumption relationships between objects, i.e., whether one class is more general than another and whether a given individual is a member of a class. Classic provides several additional reasoning techniques including inheritance, propagation, completion, and simple forward-chaining rules.

We developed a domain model of feature information by analyzing the RDDI definitions. In addition to representing permanent domain knowledge about features, options, feature clusters, etc., the model also
Shown on the left is the CD-CON Browsing Area and on the right is the CD-CON Work Area. The Browsing Area shows a graph of feature clusters, features, bound features, and options. The user has requested a search, and objects that did not satisfy the search were grayed out. The Work Area shows the current state of the Customer Delivery. The user selected the IN (Intelligent Network) application (the topmost node in the left column), which led the system to add all the basic features of that application (and any features these features required). The user added a number of other features to the delivery directly. Some of these features required other features, which were added automatically. Two of the features were inconsistent with each other (indicated by the background pattern). (CD-CON screens are in color, so a black and white image loses some information.)

Figure 1: The Customer Delivery Construction Environment
includes concepts for representing working data that changes dynamically as a Customer Delivery is constructed. Status information about each feature in the delivery is recorded that includes any other features in the delivery it is inconsistent with or requires and justifications for that feature being in the delivery. There are three types of justifications: (1) a feature was added directly by a user, (2) a feature is a required feature of another feature in the delivery, and (3) a feature is a basic or premium feature of an application being delivered. Justifications are used to perform TMS-style reasoning. We coded these reasoning routines and simple rule processing and default mechanisms in C. We discuss and illustrate the reasoning capabilities of the system in sections 3.3 and 3.5.

3.2 Browsing Area: Supporting artifact construction

The CD-CON Browsing Area and its associated functionality provide users with access to the feature knowledge base. The knowledge base is viewed as one of a set of directed graphs, each of which displays different relationships among objects in the KB. The most common view (shown in figure 1) shows feature clusters, the features in each cluster, and the bound features and options of each feature. A second view shows applications and their basic and premium features, and a third view shows features and dependencies among features.

Objects in the graph may be selected, then manipulated by various commands, some of which apply in any of the three basic view types, others of which apply only to a particular view. Commands common to all views allow the detailed investigation of particular objects, e.g., by requesting a textual description or a table of selected attributes. A command specific to the feature-dependency view allows a user to specify a new dependency between features, thus evolving the knowledge base. Because of the size and complexity of the knowledge base, search functionality is very important. A set of menus (see figure 2) lets users specify search patterns that are used to filter the objects that are displayed in the graph. By default, all objects that do not satisfy the search are dimmed; users also may choose to hide objects that do not satisfy the search. Examples of common types of searches include searching for objects whose descriptions match a specified phrase, features owned by a particular individual, or packages of features that were delivered to a certain customer. An important feature is that search is iterative: users can specify a series of searches, each of which applies to the results of the previous search.

The delivery graph contains two types of nodes – features and applications – and four types of links that represent different relationships between objects:

- a requires link connects a feature to any features it requires,
- an inconsistency link connects two features that may not be delivered together,
- a basic feature link connects an application to all its basic features, and
- a premium feature link connects an application to all its premium features.

An important convention that the layout of the Customer Delivery graph respects is that objects added by the user always appear in the leftmost column, and the top to bottom ordering reflects the temporal ordering in which objects were added.

Figure 2: Menus for specifying a search
<table>
<thead>
<tr>
<th>User action</th>
<th>Inference Type</th>
<th>Specific Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add (X)</td>
<td>Group ⇒ Member</td>
<td>if X is an application, then add its basic features as necessary, and create all necessary justifications</td>
</tr>
<tr>
<td></td>
<td>Class ⇒ Default Member</td>
<td>if X is a feature cluster, then add its features as necessary, and create all necessary justifications</td>
</tr>
<tr>
<td></td>
<td>Dependency Maintenance</td>
<td>if X is a feature, then add its default bound feature</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if X is a feature, then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for all features Y such that X requires Y, add Y to the delivery if necessary, and create the proper justification for Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if X is a feature, then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for all features Y such that X is inconsistent with Y, if Y is in the delivery, then create an inconsistency link between X and Y</td>
</tr>
<tr>
<td>Delete (X)</td>
<td>Dependency Maintenance</td>
<td>Verify that X can be deleted, i.e., that it is neither required by another feature in the delivery nor a basic feature of an application being delivered</td>
</tr>
<tr>
<td></td>
<td>Status Update</td>
<td>if X requires Y (Y in the delivery), then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if there does not exist X’ in the delivery such that X’ requires Y, then mark Y as unsupported (leaving it up to user to determine whether Y still needs to be in the delivery)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if X is inconsistent with Y (Y in the delivery), then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if there does not exist X’ in the delivery such that X’ is inconsistent with Y, then restore Y’s previous status</td>
</tr>
</tbody>
</table>

Table 1: Inferences Performed to Support the Construction Process

The Work Area provides several additional important facilities. First, it allows users to save the current delivery as a reusable package. Information such as the customer for the delivery and the user who created it are stored with the package, and the user may add textual comments. This information makes it easier to retrieve packages, thus facilitating reuse.

Second, users may load in a previous delivery, either completely replacing the current delivery, or merging it with their work to date. When two deliveries are merged, the system checks to see whether the two deliveries contain different bound features for the same feature; if so, it flags the feature and modifies its display to bring it to the user’s attention. Third, if no existing bound feature of a feature exactly satisfies the customer requirements, users can create new bound features easily, using a window that lists the options for the feature and the legal settings for each option.

### 3.4 Object-centered information delivery

A central feature of the Customer Delivery Construction Environment is its object-centered method of information delivery. Building on ideas from our own work [13, 15] and the work of others [5, 19], we have designed the system to encode task relevant information on the display of the work objects themselves. The goal is to let users stay on task, maintaining their engagement with their normal work objects rather than being forced to consult separate textual information or agendas of things-to-do. The object-centered approach is used both to deliver the assistance computed by the system and to allow users to tailor the object displays to focus on properties relevant to their current task.

First, as the system computes assistance in response to user modifications of the Customer Delivery, it may have to add objects, add relationships between objects, or modify object state information (as detailed in table 1). All these changes are reflected in the visual presentation of the delivery graph, both in the configuration of objects and in display properties such as node color and background pattern, link color, and font selection.

Second, the work objects have large quantities of associated, potentially dynamic domain information, different subsets of which are relevant for different tasks. CD-CON contains a set of default mapping rules that specify which properties of objects should be displayed and how they should be displayed. For example, in the delivery graph, inconsistent objects are colored red, and a special background pattern indicates features for which multiple bound features have been specified. Most important, an editor is provided that allows users to tailor these mappings, selecting different properties to display and/or different resources to display them.

### 3.5 Example interaction

We illustrate CD-CON’s reasoning capabilities and interaction paradigm with a simple, artificial example. Let us suppose the feature knowledge base contains the features A, B, C, D, X, and Y and the relationships A requires B, A requires C, B requires D, X requires C,
and Y inconsistent with D. We now trace through a brief scenario involving this knowledge base. We use lowercase letters to indicate default bound features, e.g., a is the default bound feature for A. "User adds feature ..." should be understood as "User requests that feature ... be added to the delivery".

1. **User adds feature A.**
   The system applies the Class \( \Rightarrow \) Default Member inference to add a to the delivery. Since A requires B and C, the system checks to see whether bound features for B and C already exist in the delivery. Since they do not, it adds b and c, and creates requirements links from a to b and a to c, and justification links from b to a and c to a. Likewise, since B requires D, the system adds d to the delivery, and creates a justification link from b to d and a justification link from d to b. When the system displays the graph, it uses italic font for b, c, and d to indicate they were added by the system.

2. **Users adds feature X.**
   The system adds x to the delivery. Since X requires C, the system checks to see whether c is in the delivery. Since it is, the system just adds a requirement link from x to c and a justification link from c to x.

3. **Users adds feature Y.**
   The system adds y to the delivery. Since Y is inconsistent with D, the system checks to see whether d is in the delivery. Since it is, an inconsistency link is created between y and d. At this point, the delivery graph displayed in the Work Area would look approximately like Figure 3, and the internal representation of the delivery (simplified for clarity) would be as follows:
   
   \[
   \begin{align*}
   a & \text{- requires: } b, c \\
   b & \text{- requires: } d \\
   c & \text{- justifications: } a \\
   d & \text{- justifications: } b \\
   \text{inconsistencies: } y \\
   x & \text{- justifications: } user \\
   y & \text{- justifications: } user \\
   \text{inconsistencies: } d
   \end{align*}
   \]

4. **User tries to delete d.**
   The system does not allow this, since d is required by b.

5. **User deletes y.**
   The system allows this deletion, marking y as deleted. (Deletion is undoable; therefore, deleted objects remain in the internal data structure, marked appropriately.) This causes it to update d's status, marking it consistent. Finally, it updates the display representations of y and d to indicate their new status.

6. **User deletes a.**
   The system marks a as deleted. This causes b to be marked as unsupported, since a was its sole justification. This in turn makes d unsupported, since b was its sole justification. Note however that c was also justified by x, so it retains its status. The system then updates the display representations of a, b, and d.

4 Related work

We discuss several representative lines of work in the areas of intelligent design support and visual information delivery and compare our research to them.

Design support environments typically provide facilities for locating and reusing components and previous designs and offer intelligent assistance in constructing a design. The COMET [8] system of Mark and colleagues at Lockheed and the "domain-oriented design environments" [7, 6, 11] of Fischer and colleagues at the University of Colorado are good examples.

COMET is based on the idea of commitment-based design. Design consists of reuse - specialization and modification - of existing design modules, with a goal of guiding current design decisions on the basis of previous decisions, thus requiring minimal changes to existing modules. The commitments of a module are the constraints that must be satisfied for it to be included in a design. The design process is highly interactive: designers locate a module that is "close" to what they need and specify changes to modules as necessary (using a restricted design language), the system retrieves existing modules that satisfy the modified description, and the designer then incorporates modules into the design, with the system assisting in satisfying commitments of all modules.

Fischer's systems include a catalog of previous designs and a construction area for building new designs. Critics compute assistance as objects are added to the design, and hypertext is used to record design rationale. Fischer's work also addresses issues in supporting collaboration between human designers centered around shared design artifacts.

As the brief discussion above shows, our work shares much with both these approaches; therefore, we simply note several differences at this point. COMET's representation of design modules is more...
The final prototype version of CD-CON has been completed. Even as a prototype, CD-CON helped to build organizational enthusiasm for the adoption of new technology, serving as a "running specification" of the behavior required in a production quality tool. We worked with members of RDDI to develop the production tool from this specification. The first version of the tool, which only implemented part of the capabilities of the prototype, was deployed in June 1994. We are continuing to collaborate with RDDI to explore ways to add more capabilities to the production version.

One undesirable aspect of our prototype is that significant parts of the reasoning capabilities are implemented in C, rather than Classic. Classic was designed deliberately with limited expressive power to ensure tractability of reasoning. However, we had several requirements that Classic did not satisfy. First, we needed a default capability. Features are represented as Classic concepts (classes), but when a user requests that a feature be added to the delivery, what really must be added is the default bound feature (an instance). Second, we needed rules that would allow us to use existential quantifiers and bind variables. We need both these capabilities to state rules that describe the dependency processing that must occur when a feature F1 is added to the delivery (F1 and F2 are features, f1 and f2 are bound features that are instances of F1 and F2, respectively):

\[
\begin{align*}
&\text{VF2 such that F1 requires F2:} \\
&\text{if } 3F2 \in F2 \text{ in the delivery} \\
&\quad \text{then add justification from f1 to f2} \\
&\text{else add default instance of F2 to delivery; call it f2} \\
&\quad \text{add justification from f1 to f2}
\end{align*}
\]

We have begun discussions with the CLASSIC designers about adding language features to support these types of inferences.

The prototype suggests several areas for interesting future work. First, for large feature lists with many dependencies, it can be difficult for a user to determine the appropriate features to delete to resolve inconsistencies. Algorithms could be developed to find the minimal set of deletions necessary to achieve overall consistency. Second, although we provide an editor that allows users to tailor how information is displayed, we currently offer no assistance in designing effective displays of information. A rule-based system that embodied the type of technique discussed by Tufte [17, 18] would be a useful addition.

Finally, we briefly discuss two general lessons we learned from this project. First, it illustrates the effectiveness of a user centered approach to the design of knowledge-based systems. Rather than developing a general representational framework, doing knowledge engineering, and developing inference routines as the first step in our project, we first spent months working to understand the organizational context, identify key user groups, and understand user needs. We next embodied these ideas in a number of paper sketches of interfaces and system interactions that we thought addressed the user needs and iterated these sketches over several more discussions with users. Finally, we
developed a domain model and reasoning routines, using the knowledge we had gained to help us in making representational decisions. We progressed from identifying user needs to developing ideas for interactive systems that would address these needs to knowledge representations and reasoning techniques.

Second, we followed what Colin Potts [10] has called an "industry as laboratory" research methodology (rather than the traditional "research then transfer" approach). A close collaboration between researchers and industrial software organizations brings benefits to both sides. Clearly, it increases the likelihood that a new technology eventually will be deployed; in our case, a system that embodies many of the ideas of the prototype has been deployed. However, perhaps less obviously, we argue that it benefits researchers, too, by giving them access to a "laboratory" in which they can apply their ideas to real problems, test solutions, and revise their ideas in light of real world feedback. In particular, it decreases the chance that the problems researchers tackle have abstracted away all the complexity that truly makes them "problems" in the real world.

6 Summary
Working in collaboration with an AT&T development organization, we have developed a system for supporting the construction of software artifacts from existing components. It enables the reuse of existing artifacts and allows users to work at a high level, with the system filling in details, maintaining consistency, and indicating areas where further work is required. Design assistance is delivered by modifying the display of work objects, allowing users to stay on task and to deal with issues when they find it convenient. A production quality version of the prototype has been developed and deployed.

Acknowledgements
We thank Doug Blewett for his assistance with Xtend and assorted systems issues. We also thank all the members of the RDDI team who have worked with us, especially Wayne Nehrlisch, Marilyn McGoorty, Pat Thomas, Jane Jesse, Tom Mack, and Delta Greene.

References