Knowledge-Based Support for the Development of Database-Centered Applications

Hany M. Atchan, Rob Bell, and Bhavani Thuraisingham

Honeywell Inc.
Corporate Systems Development Division
1000 Boone Avenue North
Golden Valley, MN 55427

Abstract

Database application productivity tools have not generally lived up to their claims for true end users. Higher productivity has been enjoyed by well-trained developers only. Several factors contribute to the skeptical attitude by end users toward productivity tools that claim to support the development of database-centered application software. The conceptual gap between what end users conceive their applications to be and what the tools require these users to do is the most important reason why end users have not strongly embraced these tools. To the end users, the application-development process is foreign and the structure used to organize the data (i.e., the details of the data model) is intimidating and arcane.

Using the Application Development Toolkit as an example, this paper illustrates that by borrowing some techniques from the artificial intelligence field, database-centered application-development productivity tools can be made more acceptable to end users and more useful to expert developers.

1. Introduction

The development of application software to manipulate data is an essential process in the use of any database management system (DBMS). The increased availability of general-purpose DBMSs has caused a strong demand for higher productivity in application software development. While query languages and data manipulation languages have improved the process of developing database-centered applications, the overall productivity increase originally thought possible simply by using these languages has not been achieved.

Application generators, report writers and forms packages have contributed to improving application software productivity but have proven to be application domain-specific and limited to trivial capabilities. Furthermore, current application generators are generally tied to one or more commercial DBMSs and, although supporting some end-user-oriented functions, are generally targeted to users with relatively high levels of programming skills.

Commercially available "productivity tools" for database-intensive application development generally lack some or all of the following capabilities:

1. A user interface simple enough to be used by nonexpert users.
2. A conceptual model of data manipulation that allows a user to express solutions using terms that are suitable to the particular application domain.
3. The capability to develop applications that are independent of their intended user interfaces.
4. The capability to identify, distinguish and search for reusable components.
5. The capability to generate applications for a variety of execution environments.
6. The capability to accommodate a variety of DBMSs and their respective data models within a common user-visible data presentation model.
7. The capability to allow advanced development by expert users—using the same development environment for customization, configuration, and version maintenance and offering flexibility in specifying the application interfaces.
8. The capability to deal with incomplete and fuzzy application specifications.
9. Good flexibility for future expansion of capabilities and support for integration of new tools with minimal impact on existing tools.

In [1] and [2] we have introduced the Application Development Toolkit (ADT) and we have shown that ADT was designed to provide the first six capabilities listed above. Reference [2] also discusses how ADT begins to support the seventh capability by redefining the development process to focus on support for customization. This paper addresses extensions to ADT to meet all the above requirements.

We are considering a development process that can yield many variations on a standard product. Clearly, configuration and version management capabilities have an important role in supporting ongoing development and maintenance of each of these variants. In particular, many dependencies can exist across customized product boundaries. Strategies for dealing with these dependencies when a change must be propagated can vary from organization to organization.
Adopting a single strategy for all is a far from optimal solution. Having more flexibility and support for expert developers is practical for the vendor.

With the tools in ADT, we have attempted to make the expression of specifications for applications as accessible as possible for inexpert users. Graphical languages and interfaces are used wherever possible. We are tracking and making available as much descriptive information as we can about the application set. However, because the specifications must be developed by the user and expressed via some kind of language, albeit graphical, we still require the user to have a clear idea of requirements before starting development if iteration of the customizations is to be avoided. To take ADT further toward the needs of the inexpert user, we must be able to start with fuzzy or partial specifications and make reasonable assumptions or recommendations as to how to complete the specification.

Most importantly, however, is that the nature of the current architecture is very rigid, with much knowledge about the development data model and development constraints being embedded in the tools themselves. A more adaptable architecture that could allow variations in both model and constraints from customization to customization would better serve the variety of customers for which any standard product must be tailored. A solution here would also enable us to add new tools to the toolkit with less impact on the tools already in use.

In essence we are suggesting that a refined development process and simplified language-based tools can go so far in supporting application development and customization by end users. (This is true for all such tools, not just ADT.) This paper describes an approach to tackling these problems by augmenting ADT with a knowledge base, accessed via an inference engine. We show how this can be done and that it will enable us to offer additional support to users, expert and untrained end user alike, and improve our capabilities for handling large numbers of customizations of a single standard product. Although we use ADT as an example, our comments are more widely applicable to productivity tools in the same general class as ADT, and based on similar technology.

2. The Application Development Toolkit

The Application Development Toolkit consists of several tools designed to support development and customization of a software product, while at the same time allowing end users to do as much customization as possible. This section briefly describes the various tools in ADT and the development process oriented to customization. (Other papers offer a more detailed description of ADT [1] and [3], and the development process [2].)

ADT supports three classes of users—end users, who use the developed applications in their everyday work; application specialists, who combine programming skills with extensive knowledge of the application domain; and system developers, with specialized expertise in database design, communications, etc., who design the core of the product to meet application domain requirements. (Note that application specialists can belong to the vendor or customer organization.) The term user can apply to any of these, as appropriate.

2.1 The Tools

The tools and their relationships are shown in Figure 1. The first three tools have graphical specification interfaces enabling end users to work with specifications directly. Each has a direct manipulation, what-you-see-is-what-you-get style [4,5], which starts from a default empty but correct specification. Operations are specified by the user selecting an object or positioning the cursor and then selecting an operation from a menu. The current specification is transformed by an operation into another correct specification. At all times the user can see a representation of the result of the current specification.

![Figure 1. ADT Architecture](image)

The Report Generator (RG) is a combination of a graphical query facility (Graphical Data Selection or GDS) and an interactive direct manipulation report format editor (RFE). The GDS specifies what data is to appear in the report and the RFE determines how it will appear.

Forms are the primary user interface mechanism for executing applications. The Forms Builder (FB) is an interactive screen painter that allows forms to be easily created or edited independently of the applications with which they may be used.

The Synthesizer (S) allows the user to pull together the pieces of the application, while ensuring that they match and the result is complete. It performs all the necessary code generation and builds makefiles to create and install an executable application, transparently to the user.

The Application Development Environment (ADE) enables the definition and use of procedural components. These are defined and used via a simple programming language, ABC [6], which supports data abstraction through Ada-like package constructs with defined interfaces and private data types. The environment supports the use of ABC with a syntax-directed editor, an internal purser and a symbolic debugger.

The Application Dictionary (AD) is a reusable component library. Whenever a package is created, the interface routines and data structures are captured in the Application Dictionary automatically. Whenever any other application component (scenario, data selection, report format, form) is created, it too is recorded, as are the combination of components into complete applications in
the Synthesizer. It tracks reusable primitives and makes them available to the user during creation of procedural logic in the SDE. It also tracks the larger application components and makes information on them available to the user via its own query interface.

The Tool Dictionary is the last subsystem. It contains essential system configuration information, for example, to define a graphic representation of this customer's unique database schema.

2.2 The Development Process

Figure 2 shows the development process for customizable application sets, as introduced in (2). Boxes represent sets of identifiable objects and arcs represent functions that transform objects into other objects. There are five sets of identifiable objects: reusable primitives, reusable components (standard and customized), and products (standard and customized). There are six functions that the model supports and that are exclusively used throughout product development and customization: primitives creation, components construction, components editing, standard application synthesis, customized application synthesis, and product inclusion/restriction.

Primitives are classified into two categories: nonalgorithmic templates and algorithmic functions. Nonalgorithmic templates are the basic starting points for constructing nonalgorithmic reusable components, e.g., empty report formats, empty data selection queries or empty forms. Algorithmic functions are procedural code made available via the ABC language.

Components are classified into two categories: nonalgorithmic specifications and algorithmic components; these are the building blocks that will be synthesized into applications. Nonalgorithmic specifications include report format components (containing report titles, page headers, group headers, data field titles, field data layout details, etc., for specific reports), data selection components (database queries), and form components (titles, field place holders, field prompts, etc., for specific forms). Algorithmic components include scenario components, which contain the overall procedural logic specification of applications, and package components, which are groupings of reusable functions.

Products are collections of applications. Applications are synthesized out of components. Applications are the basic units that provide the necessary user-identifiable functions of the final product. Standard products contain the applications that perform the core functions required by a specific application domain. Any deviation from this core functionality requires a customization effort that will result in a customized product.

3. What Else Do End Users Need?

The development of application software to manipulate data must be simplified before we can expect end users to fully participate in the development process. To accomplish this we need to provide intelligent and easy-to-use interfaces and support for end users. In the case of ADT, even though many concepts were included to encourage end users to participate in the development process, our initial experience has led us to believe that a more end-user-oriented approach is required. A more "natural," accommodating and application-oriented interface is definitely needed. The discussion below presents a characteristic set of problems that are found in the class of productivity tools similar (at least in their claim) to ADT. Later we present a series of possible improvements that shed some light on deficiencies in current state-of-the-art application-generation systems and productivity tools.

Even though ADT offers substantial on-line help and support features throughout the man-machine interface protocol, we have identified some deficiencies that are common among all productivity tools.

3.1 The Development Process

Productivity tools usually support a general development process that must be closely followed. It consists of various steps or operations which must be performed in a specific sequence. An objective examination of the rationale for such a process shows it to be an artificial road map that has little to do with the nature of the problem. Furthermore, the major purpose of this road map is to produce the set of objects required by the computer to perform the prescribed solution. As a simple example, consider a DBMS that provides an on-line SQL facility. A simplified process in this case consists of three steps: typing the query text in SQL, giving the query a unique name, and saving it to a file. To an end user it is simpler to have an interface that automatically "remembers" all that he does, and only "forgets" the queries that he specifically recognizes as "wrong." Furthermore, an end user usually "remembers" the query by the information it provides him. To ask the end user to give the query a unique identifier (probably the file name where the query is stored) and to specifically tell the computer to save this query is an artificial constraint that the creator of the DBMS SQL facility has imposed on the end user. It is an unnatural process that the end user is asked to perform and therefore the whole process is hard to understand. The real reason that the system requires the end user to save the query for later use is to limit the amount of computer memory used. The system requires the end user to name the query so that it can be identified quickly later when the user wants to use it again. In short, the need to hide memory size and
speed of identification limitations makes this example process unnatural and hard to understand.

In ADT, the development process specifically provides for easy customization of applications after they have been developed. In fact, the high-level process to develop applications in ADT consists of building the set of necessary components (either from empty templates or by customizing existing components), and synthesizing these components into the final (reporting or nonreporting) application. As described earlier, the synthesis step allows the ADT user a powerful mechanism to customize the details of the interface of the application, for example, without having to dig in the code to find the proper places for change. The integration details required to put the application back together are taken care of by the synthesizer. Here flexibility is gained by introducing an extra step (the synthesis) that is new to expert developers and programmers!

This example from ADT is really representative of many other productivity tools where, in order to accommodate a large audience, the system has to become more complicated and the resulting process is hard to understand. A trade-off exists between flexibility and power on one hand and ease of use on the other hand. Although it seems that this trade-off forces shortcomings to be built into every productivity tool, we believe that by using some artificial intelligence techniques (7,8) the process can be made more powerful for expert developers and more friendly to end users.

3.2 The Application Dictionary's Data Model

By examining the underlying structure of data internally maintained by these productivity tools to support the development process one can easily see the influence of this structure on the user-machine interface protocol. As an example, consider ADT where the internal structure of the data is relational and where a flexible report generator allows the user to develop the query independently from the report format. In this case flexibility is definitely improved since the same query can be coupled with different report scripts to create different reports (with the same data). By the same token, ADT also allows a report script to be coupled with different queries to present different data in "similar-looking" reports. For ADT to provide such flexibility (which is definitely useful) the internal structure of the data has to allow a many-to-many association scheme between the queries and the report formats. To identify these associations, the system needs to impose a restriction that the user provide a unique identification of the association to which he is referring. This is a price that users have to pay in order to gain the flexibility described above. If, however, a certain user does not want such flexibility but wants all the other features of ADT, this scheme of identification is unjustifiable and unnatural. In this case, the need to cover up for the rigidity of the "relational" structure of the data internally kept by ADT has added unnecessary complexity for the end user.

To allow a wide range of possibilities, the Application Dictionary in ADT had to include many component types and classes. The maintenance of the consistency of these components is another trade-off that faced us. If too many assumptions are made (and therefore a lot of restrictions are enforced by the system) performance would diminish and the flexibility of the possible operations would be severely hindered. If, on the other hand, too few restrictions are enforced by the system its flexibility would be enhanced but the complexity of the user's task would be increased. This is because whatever restrictions need to be enforced, but are not enforced by ADT, have to be entrusted to the user.

Using a rigid data model to support the internal operations of ADT is a handicap. To allow flexibility without having to trust users to enforce the restrictions by themselves requires a more flexible data model. We have studied some knowledge maintenance systems that use semantic data models (9,10) and feel that using knowledge base support systems with semantic or hyper-semantic (11) data models can help address these problems.

3.3 Better Support for End Users

Existing artificial intelligence technology offers many possibilities that support better benefit for end users from productivity tools. The most obvious technology is natural language processing in interpreting syntax-free English specification of data retrieval queries. Although this is a problem that has been solved in existing commercial database products, the user is required to know the structure of the data he is trying to access. It seems that what is currently available is only a partial solution. There are, however, other examples where the technology is ready to provide the required results.

The remainder of this section presents brief descriptions of representative examples of problems that can be solved today and that would greatly improve the usability of productivity tools by end users. In the next section we present the details of a knowledge representation model that could be employed to structure the data internally kept by ADT. In Section 5 we present solutions, described using the model of Section 4, to these example problems.

Case A: Guidance for End Users Through the Development Process. The concept of transaction has never been clearly established for productivity tools. We define a transaction to be a session where an end user performs a series of operations identifiable by the system. A transaction starts when the end user requests the help of ADT in developing an application (after he identifies himself to the system as a registered user) and ends when the user asks to cancel the current transaction or has successfully produced a complete and executable application. We assume that the system keeps track of the operations performed by the end user within the current transaction. The problem: Given the state of the current transaction, can the system tell the end user what his next operation should be?

Case B: Guidance for End Users or Expert Developers Through a Customization Task. When users or developers put several components into an application, they are linked by an imaginary dependency relationship that form a logical dependency graph. Several dependency graphs might be connected through common
components. Customizing (or modifying) one component in this graph might affect the other applications that use this component. The problem: Given a dependency graph and a specific customization task, can the system prevent (or at least detect and warn against) the side effects potentially caused by this customization?

Case C: Intelligent Support for Formulation of Data Access Queries. All the commercially available DBMSs use one data model or another. All such data models force a structure on the modeled data, which is foreign to end users. Furthermore, the naming conventions of data structures and their fields is another foreign concept. End users recognize the data by the meaning that such data carry to them and think in terms of examples. Artificial structures used to model relationships between various data elements do not fit well in the end users' frame of reference. Even Query-By-Example (QBE) requires familiarity with the structure of the data and therefore requires learning of unnatural constructs on the part of end users. The problem: By accepting example values from end users and without requiring any knowledge of the structure or the nomenclature of data, can the system formulate an accurate specification of the end user's query?

4. Knowledge Representation

The model we need for the application under consideration should capture not only the entities and relationships between the entities but also other aspects of the real world, such as operations, constraints, temporal restrictions and heuristic rules. Therefore, the model we chose to represent the knowledge needed for ADT is a frame/object-based model, which is a combination of the object-based model, slot and filler representations, and functional approaches discussed in [13], [14], and [15]. Our model can also be classified as a hyper-semantic model [11] as it offers extensions to the semantic network data model [9]. It also displays some other characteristics discussed in [16].

The characteristics of our model include:

- Generalization, where similar object types are abstracted into a higher level object type,
- Classification, where specific instances are considered as a higher level object type,
- Aggregation, where an object is related to its components,
- Membership, where several object types are considered as a higher level set object type,
- Inheritance of properties along a class hierarchy.

The notion of an object is the basic building block in our model. An object can be used to represent a simple entity, such as an integer, or a complex entity, such as some real-world object. That is, any entity, relationship, rule or policy can be described by the notion of an object.

Specific rules that are of interest to us are the following:

- Constraint Rules: These rules enforce restrictions on one or more objects. They could depend on content, context or the number of objects under consideration. The type of constraints include integrity and security constraints.
- Inference Rules: These rules provide the mechanism to deduce new information from existing information. Inferencing capability is essential in any knowledge-based system because the data/knowledge handled is very large and it is not feasible to specify all of it [10].
- Heuristic Rules [17]: These rules provide an information derivation mechanism when uncertain and incomplete information is present. In a knowledge-based system, some required information may not be available in advance; that is, information may only be available in stages. Therefore the system should be able to reason with partial data/knowledge. The heuristic rules provide for the facility to do this and help in decision making.
- Temporal Rules: These rules relate object types by synchronous or asynchronous characteristics and are considered as a higher level object type.

Section 5 describes how the knowledge represented by this model is used by the inference engine component of the knowledge-based system.

5. The Smarter Toolkit

The current ADT internally maintains an extensive body of knowledge basically structured in a relational fashion. As we have shown above, this restricts the amount of freedom that end users have in specifying their requirements. By using a more flexible knowledge representation model (like the one described above) with many powerful constructs, ADT can better support end users and expert users too. Using such a model would make it possible to represent the internal data in the same way meta data and the rules that govern the maintenance of this data are maintained. Furthermore, this allows the data, the meta data and the maintenance rules to be modifiable. This is the edge needed to increase power and flexibility while at the same time increasing suitability to end users.

5.1 ADT's Architecture Revisited

The objective of the revisited architecture is to provide a platform where the same capabilities of ADT can be provided but where the interface can also be handled in a flexible and intelligent way. Figure 3 shows the new architecture in which we added two subsystems that were not included in the old architecture: the Inference Engine (IE) and the Development Process Guide (DProv). The Application Dictionary and the Tool Dictionary (TD) remain as before except that in the new architecture they are contained in a flexible hyper-semantic model-based Knowledge and Rule Base (KRB). This KRB is extended to contain additional information not contained in the original AD and TD. The contents of the knowledge base are of two types: information that is manipulated solely by the tools (regarding transactions, components, etc.) and information that pertains to the setup of the ADT for the specific
customer environment. The contents of the rule base (which is integrated with the knowledge base) are of three types: temporal (defining time precedence and survivability that govern activities, operations, entities, etc.), constraints (defining conditions on integrity of entities, objects, etc.) and heuristic (defining strategies for searching through the rule base or the knowledge base, browsing through network values, guessing routes in the process steps network, etc.).

Figure 3. New Architecture for ADT

In the new architecture, the DPGuide is the common interface that allows all users to access other subsystems through a uniform interface style. This also allows ADT to handle access rights, transactions, and operation modes in a similar fashion. The DPGuide provides several modes of support to the user. In the expert user mode, the system provides very little guidance in the process (except if specifically requested). In the fully guided mode, the system assumes an end user with no experience and therefore provides total tracking of the user transaction to guide the user when the system feels he is confused or in trouble. Several support modes can be provided between these two modes. Whenever ADT is set up for a specific customer, a knowledge engineer helps customize the knowledge structure and rule base for the customer through a special interface provided by the DPGuide.

The Inference Engine is aware of all access to KRB and has the ability to exercise complete control over it. In the expert user mode, IE’s interference is kept to a minimum. However, IE is still in charge of verifying all accesses to the knowledge base to fire any enabled triggers and to help the various tools update the knowledge and rule base. The knowledge of how to update AD and TD used to be embedded in the various tools that access these dictionaries. In the new architecture, this is formally provided by the rules, which IE can handle. The tools do not have to be dependent on the internal structure of the dictionaries.

5.2 Using the Knowledge and Rule Base

To illustrate how the KRB can be used to improve the facilities offered by ADT, in this section we present the solutions to the hypothetical problems presented in Section 3 (Cases A, B and C). We demonstrate the uses of each of the constructs and abstraction mechanisms presented in the model of Section 4. We also make many assumptions as we proceed in our illustrations. The solutions are each applied to a miniature knowledge base that is specifically "trimmed to contain only the important concepts. The example rules are also chosen to illustrate the important issues, and therefore a few unimportant details are left to the reader’s imagination.

Case A: Guidance for End Users Through the Development Process. Figure 4 presents the knowledge representation structure for application building. An application can be either a reporting or a nonreporting application (this is a generalization). A nonreporting application is an aggregation (represented by a circle) of any number of forms and exactly one scenario. The same scenario can appear in more than one application. The same is true for forms. Between a scenario and packages there is a used-in aggregation where any number of packages can be used-in a scenario. Similar interpretations can be given to the reporting branch of the structure shown. Notice that two different aggregations link forms to reporting and nonreporting applications and that instantiations are not shown in this diagram. Details about instantiations are discussed in the solution for Case B.

Figure 4. Example Knowledge and Rule Base for Case A

Figure 4 also shows temporal and constraint rules as dark arrows pointing to the relevant entities. For our example, names of temporal rules and constraint rules start with “T” and “C,” respectively. These rules are to be applied within a transaction (as defined in Section 3, Case A above).

T1 is a rule specifying that before an application can be recognized, T2 or T3 has to be satisfied. T3 specifies that before a nonreporting application can be recognized, T4 has to be satisfied. T4 specifies that it is satisfied if C3 is respected. C3 is a constraint rule that checks the existence and identification of the scenario in question. C3 also checks that if the scenario requires one or more forms, then one or more forms exist and are identified within the current transaction. If the scenario needs to be built, then constraint rule C1 specifies the condition of integrity of the scenario (for example, no compilation errors, and C2 is satisfied). C2 is a constraint on the used-in aggregation; it
requires that if one or more packages are used in the scenario, then they need to exist in the knowledge base.

To solve the problem of guiding the end user through the development process, DPGuide knows that the objective of the development process is to produce a complete application. (This knowledge does not have to be built in the DPGuide; it could be a rule included in KRB, too.) By checking the class of application, DPGuide sees the need to satisfy rule T1. With the exception of rule T1, it should be easy for the reader to see how the IE can cascade through the rules presented above to figure out the first unsatisfied rule and return the result to DPGuide, which in turn can direct the end user to satisfy it (by translating the returned result into the operation that the user should perform next; this translation can be done by inferencing on some different part of KRB). T1 contains an alternative path (T2 or T3). This allows us to show how heuristic rules can be used. Assume H1 is a rule that applies to the execution of rule T1 (this is not shown in Figure 4). This rule H1 tells IE that a choice has to be made and how to make it. Two cases arise: This information could already have been given by the end user and maintained by DPGuide, or this information is not known. H1 can be as simple as directing DPGuide to ask the user to make a specific choice. Alternatively, it could be more complex (like checking the history of transactions for the specific end user, checking other recommendations entered as rules by the system administrator, using some probability measure to guess first and then confirm the guess with the user, etc.).

In short, a knowledge and rule representation scheme such as the one described here can allow a multitude of options that are simply not possible using conventional, static-schema support systems.

Case B: Guidance for End Users or Expert Developers Through a Customization Task. Figure 5 shows the instantiations dependency graph that can be used to solve the Case B problem. The presented knowledge only covers part of what was shown in Figure 4. The instantiations are indicated by the term ‘ins’ on the connecting arc. For example, S1 is an instance of the scenario class; P1, P2, and P3 are instances of the package class; and B1 is an instance of the used-in aggregation class B (that tells which packages are used in each scenario).

As Figure 5 shows, P3 has been used in S1 and S2. If a user attempts to modify P3 to satisfy a customization task, the other aggregation using P3 might become inconsistent. The capability to detect a possible conflict has been addressed in many previous works, including [18]. For the existing systems that provide this capability, handling such a conflict after it has been detected has always been built into the system. In the new ADT architecture, the mechanism needed for handling this conflict can be flexibly included as a heuristic rule that is consulted by IE whenever it occurs and needs to be resolved. This way, the knowledge engineer, in consultation with the specific customer, might decide that users should be refused the right to modify instances that might create such a conflict. Alternatively, the knowledge engineer can include a rule that directs IE to create a new version of the instance in question automatically (and link it to the proper aggregation transparently to the user) as a solution to the conflict.

In short, by specifically identifying and isolating the knowledge required to resolve conflicts, ADT can provide the necessary flexibility to handle a variety of conflict-resolution schemes, rather than exclusively supporting one scheme that might be useful to some users but not others.

Case C: Intelligent Support for Formulation of Data Access Queries. In ADT and most existing systems that provide some fourth-generation language querying capability, some knowledge of the structure of the data is usually required (even query-by-example systems force a restricted syntax to be used in conjunction with the familiarity with the data structures in question). That is why guessing a data access query by allowing end users to specify examples of the information they are trying to access is usually difficult. Nevertheless, there are cases where some patterns could be devised that allow educated guesses of data access queries based on the provided data examples.

The problem usually lies in matching the provided data example to the target structure of the data. Consider for example a simple relational parts-supplier-city database. If a user requests “information about Extarbruhn,” the system would need to know in which relation it should look for Extarbruhn. Further it would need to know if Extarbruhn is a specific value of a key field or some other field.

We do not claim that ADT will be able to solve this problem. However, by using some heuristic rules, the system might be able to infer or guess at a proper query that the user has in mind. For example, if the user is a warehouse employee, it is very likely that the user wants to know about the current stock of the part called Extarbruhn. In the absence of a computer solution, a real situation like this could be easily solved by a person who knows that this user is only interested in parts. Therefore this knowledge is a helpful heuristic (or at least a hint) that would help resolve this specific request for “information about Extarbruhn.” By providing the capability to capture some of the knowledge not usually included in static-schema support systems, IE can use heuristic rules to formulate the correct queries (at least with a degree of certainty). In a true environment using this scheme of end-user support, extensive work might be required by the
knowledge engineer to extract the necessary knowledge. If this support is important to the customer, the system should at least allow for it.

One final note about this type of support. We recognize that the system cannot always arrive at a guaranteed correct query formulation. Sometimes a 50 percent accurate guess is all that the system can offer. Even in these cases, the system can offer its findings to the end user in some sort of a descriptive statement and let the user decide for himself. In other words, even if an inferencing mechanism such as the one described above does not always produce accurate query formulations, sometimes an intelligent guess might help. After all, this is the same thing an expert person would do if he had to solve the same problem!

6. Conclusion

In the last few years the enthusiasm about application-generation systems has decreased considerably. The reason is not so much that all the problems have been solved, but rather that end users did not feel that these systems are targeted to them (as is claimed by most of these systems).

Modern application productivity tools and application generators have to provide extensive support for end users before they can realize the full benefits of these tools. It is very promising, as we have shown in this paper, to use artificial intelligence techniques to compensate for artificial modelin, structures and development processes of existing productivity tools. By doing so, the end users will at least feel comfortable using these tools because they will benefit from an intelligent user interface and capabilities.

In the past year we have completed the implementation of a working ADT. In this paper we have presented a preview of a new generation of productivity tools (like ADT) that are specifically designed to allow the active participation of end users.

References


*These references are Honeywell proprietary.