USING ANNOTATED CONCEPTUAL MODELS TO DERIVE INFORMATION SYSTEM IMPLEMENTATIONS

Anthony Berglas
The University of Queensland
Brisbane 4072 Australia
berglas@cs.uq.edu.au.

ABSTRACT

Producing production quality information systems from conceptual descriptions is a time consuming process that employs many of the world's programmers. Although most of this programming is fairly routine, the process has not been amenable to simple automation because conceptual models do not provide sufficient parameters to make all the implementation decisions that are required, and numerous special cases arise in practice. Most commercial CASE tools address these problems by essentially implementing a waterfall model in which the development proceeds from analysis through design, layout and coding phases in a partially automated manner, but the analyst/programmer must heavily edit each intermediate stage.

This paper demonstrates that by recognising the nature of *information systems*, it is possible to specify applications completely using a conceptual model that has been annotated with additional parameters that guide automated implementation. More importantly, it will be argued that a manageable number of annotations are sufficient to implement realistic applications, and techniques will be described that enabled the author's commercial CASE tool, the *Intelligent Develope* to automated implementation without requiring complex theorem proving technology.

Keywords: Information Systems, Conceptual Models, CASE, Extended NIAM, Reuse, User Interface Design

INTRODUCTION

There has been considerable research into the development of sophisticated techniques that capture abstract requirements and refine them into functional prototypes. Good examples include the programmer's apprentice Reubenstein (1991) which is concerned with resolving informal descriptions that are ambiguous, inconsistent and incomplete by referencing a library of reusable clichés, and the Daida project Jarke (1992) which uses advanced knowledge representation techniques (Telos) to capture general requirements and map them to conceptual and implementation descriptions. Daida also uses a general system of non-functional goals to determine optimum implementations. Both of these systems describe transactions using first order logic which requires advanced theorem proving technology.

While designing detailed conceptual models may require human like intelligence that is difficult to reproduce, much of the cost of developing applications using conventional CASE technology is incurred in the writing and maintenance of 4GL code and end user documentation that implements these models. Tools such as Oracle (1993), or described in Ovum (1992) effectively implement a waterfall model in which tools partially automate the mapping from conceptual to logical models and then 4GL code, but the programmer/analyst must extensively edit each intermediate stage. This is an expensive process and makes the resultant applications difficult to maintain. Object oriented interface tools such as Borras (1992), Flynn (1992), and King (1993) facilitate more complex graphical manipulation of objects, but still required programmers to essentially edit generated code.

This paper argues that by recognizing the limited types of operations that are commonly performed by conventional *information systems,* it is possible to largely automate the implementation of production quality applications from conceptual models that contain a manageable number of annotations. Although the emphasis of this paper is on how applications are technically implemented, automating implementation can also be used to empower more sophisticated end users to usefully tailor conceptual models of large systems in a controlled manner.

The next section describes the types of decisions that need to be made, how rules can be used to automate them, and mechanisms to handle exceptions. The IFIP conference example Ifip (1982) will then be used to give examples of specific annotations. Techniques for working with annotated models will be discussed including the possibility of enabling end users to tailor conceptual models.

DECISIONS, RULES AND ANNOTATIONS

The decisions that are required to implement information systems may be classified in terms of where they would be made in a waterfall model, which include:-

- Generating Relational Schemas: When denormalizations should be introduced; whether derived facts (attributes) should be evaluated eagerly or lazily; whether constraints should be maintained in the applications or an active database; how data should be distributed over a network.
- Outlining User Interfaces: How many user interfaces there should be and the scope of each one; determining which attributes should be manipulated together; controlling end user access to sensitive data.
- Detailing User Interfaces: How many windows should be used; how should the fields be laid out within each window; which widgets[1] should be used for each field; how can non-form interfaces be used.
- 4GL Coding: How to code unclassified business rules and special cases.

Many of these decisions can be made on the basis of well defined rules. For example, the decision to select an appropriate widget largely depends on the number of valid values a field may have — a "ding bat" is good for two alternatives, "radio buttons" for up to four, thereafter a conventional text field is more appropriate. Another rule states that when a screen includes a foreign key, the attribute(s) that identify the foreign entity to a user (ie. the *Descriptive* ones) should also be displayed if there is sufficient room. For example, students' names would normally be displayed next to their number on a class list, rather than their age, say.

At the relational level, the use of keyword tables to allow efficient searches for words within strings is a common reason to introduce denormalizations[2]. The decision to use one could be made by the following rule:-

If a column is used for sub-word searching,
  And the number of rows is > = 500
  And searching is at least four times more common than updating,
  And this column is often used as the sole search criteria
Then create a keyword table.

Conceptual schemas and dynamic models provide the base parameters required by the rules. Some parameters such as an attribute's data type form part of most modeling techniques while others may be introduced to the models using additional *Annotations*. Typical annotations indicate which attributes *Describe* foreign keys, how and when derived attributes are calculated, and when keyword searching is useful.

Annotations can be loosely classified in terms of their "conceptual purity" as follows:-

- Conceptual Constraints that are part of a NIAM model, such as uniqueness and exclusion constraints.
- Structured business rules that enforce more complex constraints.
- Conceptual Parameters that are independent of specific implementation techniques, but do not directly affect conceptual semantics. Examples include specifying which attributes are *Descriptive*, which uniqueness constraint defines the primary key, or whether a subtype is collapsible[3].
- Overrides to the standard derivation rules. These can be quite specific, for example, to specify that a particular attribute on a specific interface should be represented using radio buttons rather than the default ding bat.
- Ad hoc 4GL Code attached to interfaces.

This classification is orthogonal to the waterfall approach used for decisions, and the more conceptual annotations may be used by several decisions. For example, an enumeration of the valid values an attribute may contain requires an integrity constraint to be created during table definition while also affecting the widget used in the user interface.

---

1. Ie. Low level user interaction mechanisms, eg. Scroll Bars and Radio Buttons.

2. Eg. if a column contains "Peter John Smith" the corresponding keyword table would have three indexed rows for "Peter", "John" and "Smith".]

3. Ie. stored in the same table as its supertype.

While implementation rules can produce useful prototypes directly from Conceptual Constraints and Parameters, Override and Ad Hoc Code annotations enable analyst/programmers to override rules or to add arbitrary procedural extensions when developing production quality applications. Unusual modules could require numerous annotations[4], but in practice most modules either require no annotations or have specific, minor idiosyncrasies. Specifying these idiosyncrasies with impure annotations at the conceptual level is undesirable, but when the need arises it is much easier to maintain a few lines of ad hoc annotations in a conceptual model than to find and understand the few changed lines interspersed amongst hundreds of lines of generated 4GL code produced by a waterfall approach. Moreover, if the schema changes in ways unrelated to the overrides, the complete 4GL program can be easily regenerated[5].

The effectiveness of this approach is dependent upon the ability of a manageable number of conceptual annotations to provide significant automation. If new special case annotations were required to effectively implement each new application module, the problem of designing a conceptual model would degenerate into that of understanding the meaning of innumerable different annotations.

The author's experience with the *Intelligent Developer* (ID) Berglas (1993b), BHA 1992, a commercial 4GL generator, shows that the following types of requirements account for the majority of 4GL code:-

- Enabling rows to be selected, edited and reported within the basic relational structure. This is usually achieved through forms oriented interfaces that process related tables, with most of each form's fields corresponding directly to columns in the database. Complications include using non-standard indexing systems such as keyword tables, and looking up *Descriptive* values for foreign keys.
- Making salient derived data conveniently available, such as totals of detail rows. This is similar to the functionality provided by conventional views, but there are subtleties such as deciding whether to use lazy or eager evaluation, and if eager the use of delta rules[6], or perhaps the derived data is a default that can be overridden.
- Different users have different access to sensitive data. This is also similar to views whose queries can relate the accessed data to user profiles, but additional details need to be addressed, such as telling a user that they have no access rather than just pretending a row does not exist.
- Invoking structured business rules.
- Producing elegant user interfaces. This involves connecting windows together, providing convenient default values, context sensitive help, and designing reasonable form layouts.

The types of requirements are limited because most of the processing involved in an *information system* can be described in terms of accessing, updating or disseminating the information that they contain using standard algorithms contained within a database manager. Most constraints on the data are of a standard type, such as referential integrity and uniqueness, while simple finite state machines can be used to provide dynamic constraints and trigger actions upon state transitions. The relatively small number of business rules that cannot be expressed as constraints to conceptual schemas account for a very small fraction of the 4GL code used.

The dozen specific types of annotations described in the following sections account for most individual annotations used in practice because they correspond to these requirements. Further, the data centred approach makes it unnecessary to detail many transactions such as "Enroll a new student" if they mainly involve accessing and editing data subject to constraints, in this case editing details about each student. By capturing common constraints and implying basic transactions, annotations reduce the need for formal specifications using languages like TaxisDL (part of Daida). It is acceptable to specify remaining ad hoc definitions procedurally because there are not many of them in an *information system*, so the limitations of technologies that automatically refine formal specifications into production quality applications can be circumvented.

This technology is inappropriate for applications that contain many operations that do not fit the model of updating a structured database subject to relatively simple rules. For example, an application where most transactions involved complicated procedural interfaces to other systems rather than just

---

[4]. Techniques for managing unusual modules will be discussed below.

[5]. A warning should be issued if this independence cannot be shown..

[6]. Eg. it is not necessary to completely recalculate a total if one detail record is changed.

updating a local database would be unsuitable — indeed a language like C++ might be more appropriate than a 4GL for such a system.

THE IFIP EXAMPLE

The IFIP example defines an information system for managing conferences. Authors submit papers, which may have been invited to conferences. The information system should keep track of papers and authors, allocate the referees, produce exception reports of overdue papers etc.
A typical Intelligent Developer style user interface is shown in figure 1, which could be used to submit papers. The user enters the Paper Nr of an existing paper, or creates a new paper record by allowing it to default to the next available number. When the ID of the conference being submitted to is entered the conference Name is shown to the user. The paper's Title, Subjects and Authors may then be entered, the latter in scrolling regions. An acknowledgment is sent to the first author once the paper has been logged.

Figure 1: Window for submitting papers to a conference.

If the user does not know the Conference ID, they may press the "Select" key to pop up the conference selection window shown in figure 2 . Conferences may then be selected by entering a fragment of a conference name or a period during which one may have been held. Most ID forms are variations of either either maintenance forms (like figure 1) or selection ones, but although these interfaces are fairly simple and have a common structure, they take hundreds of lines to implement properly in a conventional 4GL

Figure 2: Window for selecting conferences

Figure 3 shows part of the corresponding NIAM or Object Role Modeling conceptual schema Halpin (1991). *Entities* are circled, and *facts*, the relationship between them, are represented as rectangles. The bar next to a square indicates the *role*(s) that identify each fact. Thus many people may work for

an institution, but each person may only work for one institution (at a time); while a paper may have several authors, each of whom may write several papers.

NIAM is similar to Entity Relationship Attribute (ERA) models, but ERA "Attributes" and "Relationships" are considered to be just a type of fact in NIAM. For example, Conference is an ERA Entity with Attributes ID, Name and Date, while Paper-has-Conference is a Relationship because Conference has Attributes. CASE tools can provide convenient ERA abstractions of NIAM schemas, and may also suppress non-conceptual annotations.

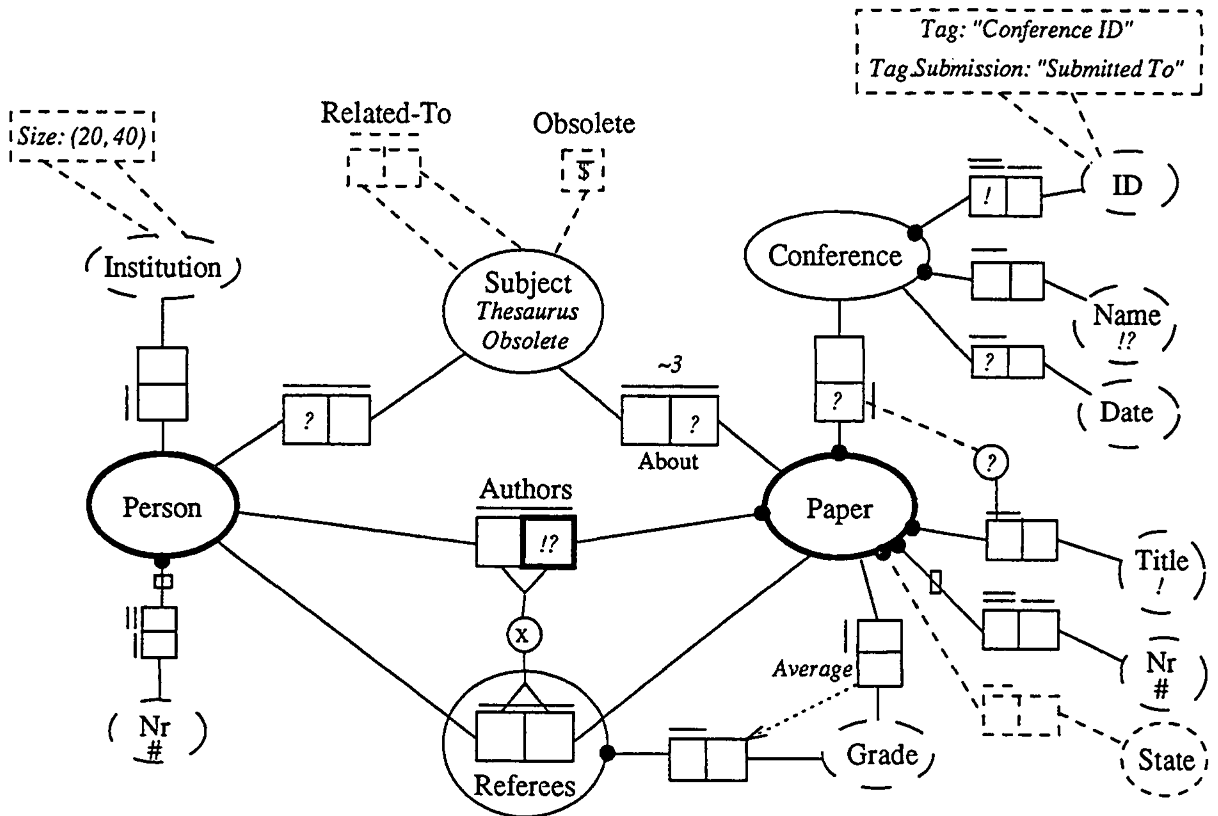

Figure 3: The IFIP Conference Conceptual Schema.

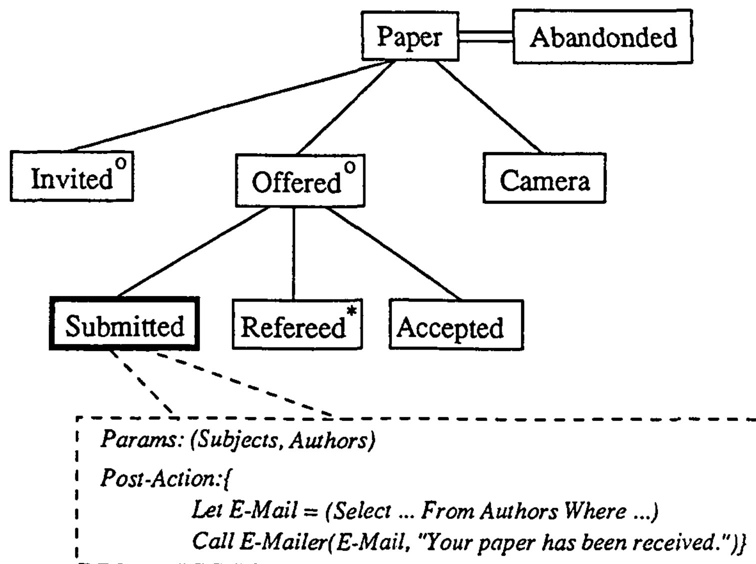

Figure 4: The Paper JSD-Like Entity Life Cycle.

Figure 4 shows the life cycle of a Paper in a JSD style notation Jackson (1984). Papers are either Invited or Offered by the authors, in which case they are Submitted, Refereed a number of times and possibly Accepted before the Camera ready copy is printed; but this could be abandoned at any stage. As no predicates have been specified to define what state a paper is currently in, an implicit State fact is created on the NIAM schema. NIAM entities can have several independent life cycles, and two state life cycles with explicit predicates encompass active database "When/If/Then" rules.

EXAMPLES OF SPECIFIC ANNOTATIONS

Table 1 lists common annotations, some of which were used in figure 3. Many of these annotations describe the conceptual models in ways that do not strictly constrain the data and so cannot be naturally described using first order logic. However, they provide important information for the production of effective implementations, especially user interfaces. For example, "~ 3" indicates that Papers *typically* concern three subjects but does not prevent some papers from referring to five subjects. This can be used to determine whether indexes should be created and the number of records to display in an interface's scrolling region[7]. All conceptual objects (Facts, Entities, Constraints and States) may be annotated, with facts inheriting annotations from entities.

Design rules can determine the purpose and basic structure of each user interface by examining an ERA abstraction of the NIAM diagram[8]. By default, each ERA Entity has a corresponding form to view and update it that contains a field for each attribute, foreign key and any associated descriptive facts. If the entity was embolded (ie. Paper and Person) a scrolling region is created for each referencing foreign key (eg. About-Subjects and Authors). Selection forms and simple reports are created in a similar mannerThe Paper Submission form in figure 1 was produced via a different mechanism, namely to provide a special purpose interface for transition to the Submitted state because it is embolded in the dynamic model. The annotation indicates that Subjects and Authors (but not Referees) are involved and how the acknowledgment should be sent. The details about each author may be edited directly on Paper maintenance forms because the Author role is embolded, while only the reference to Subjects is shown. Note that most of the interface's functionality concerns the editing of information about papers, even though it implements the *transaction* of logging new papers.

Annotations are used to *describe* user interface layouts rather than specifying them explicitly using a screen painter. Screen painting involves using a pretty interface to issue very low level commands such as "place field X at (12, 34)" rather than "place field X *near* field Y". This makes it difficult to generate new layouts when the underlying schema changes, and curtails the interesting option of generating layouts at run time in response to the subtype instances being manipulated or the end user resizing graphical windows. Screen painting is also unsuitable for non-forms interfaces such as node/arc graphs or Gantt charts.

Therefore details of each layout should be determined by an algorithm, with user interface annotations providing hints such as which facts are related or partial orders of fields. These gentle annotations will usually be applicable for all interfaces that reference a fact. For example, a report will probably want fields to be laid out in the same order as an interactive interface even though their actual positions may be quite different. The Intelligent Developer's simple layout algorithm is adequate for simple interfaces, while techniques such as Wiecha (1990) and Pizano (1993) could provide greater automation.

Annotations may be addressed to specific interfaces when the need arises by using different "facets". Thus "Conference ID" has been specified as the boiler plate tag in figure 3[9] for all interfaces except Paper Submission, for which it has been specialized to "Submitted To". As the conceptual model is used as a complete specification, it also becomes desirable to be able to restrict annotations to specific subtypes or dynamic states. For example, it might be mandatory for all Papers to be refereed, except invited ones. This is orthogonal to facet specialization.

---

7. This annotation could have been deduced by examining the populations of live databases, but affects implementation like other annotations and can aid an analyst's understanding of a schema.

8. This brief description only provides an approximation to the complete algorithm and ignores subtypes, problems with large forms, keys generated from the data etc.

9. The tag defaults to the fact name (eg. "Name") in the Intelligent Developer, unless it is specifically "ID", "Nr" or "Code" in which case it is prepended by table name, ie. "Conference ID" is actually the default.

Annotations can also improve the behaviour of other database interfaces because they usually describe the nature of the schema itself. For example, QBE[10] could use a *descriptive* annotation to automatically join Person.Name by default whenever the Author table is queried rather than just showing the uninformative Person.Nr, and the SQL code required to calculate derived facts ("*") could be incorporated into queries as required.

Berglas (1994) argues that the definitions of active business rules should be structured in terms of the conceptual schema using KL-ONE Brackmann 1985 style classification and method combination. Thus there are annotations

Symbol	Meaning
	Conceptual Constraints
•	A mandatory role, ie. every entity instance must have a corresponding fact instance.
ⓧ	Exclusion constraint eg. one cannot referee ones own paper.
□	Fixed value, fact instances cannot be changed once created (not standard NIAM).
	Conceptual Parameters
═	Primary key (always unique).
!	Descriptive, eg. The conference Name is shown in figure 1 rather than its Date (say), although the descriptive ID (eg. "CIKM93") would suffice (cf. Person-Nr "92364").
?	Selective, so is useful for a selection criteria (eg. in figure 2) and suggests an index could be created.
Ⓟ	Selective fact usually used in combination with others. This affects user interfaces, and may create multi-column indexes.
$\sim n$	Typical populations, eg. there are *usually* 3 Subjects per Paper.
o	A role that is usually null. Affects user interfaces and database sizing.
×	Exclude this fact from a (specific) interface.
Size	Typical and maximum sizes of each field/column.
...	Not all instances are stored ie. they may be purged.
	User Interface Options
Embolding	Affects user interface structure, described below.
Tag	Fixed boiler plate text on forms and reports.
&X	In the fact group named X, should be displayed close to other members of the group.
>Afact	Defines a partial ordering of fields in user interfaces.
WSize	Actual, fixed size to use for a widget.
Offset	(Vertical, Horizontal). Explicit position in a window relative to previous field.
Widget	Name of the widget to use, possibly with specific parameters.
Popup	Name of a pop up program, eg. Conference Selection.
	Non-Standard Business Rules
*	Lazy derived fact (the actual 4GL code is shown on request).
$\overline{*}$	Eager derived fact (stored in the database).
+	Default values provided for a fact instance if it was Null.
$\overline{+}$	Default value calculated once upon a state transition and stored.
⊙	Conditions that determine when an instance belongs to a subtype.
⊗	Database action to performed when an entity is (re)classified.
⊕	Add hoc 4GL code to be hacked into Forms interfaces that contain the corresponding field or table.
⊖	Like ⊕ but only affects the user interface.
	Schema Management
△	User configurable conceptual object.
▽	Conceptual object owned by an end user.
†, ‡	(Very) "important" object for schema abstraction.
	Conceptual Library Invocation
Σ	Total of pointed fact, a special case of *. Likewise $\overline{\Sigma}$, *Average*, *Count*, *Max*, etc.
#	Automatic key generation, eg. give Papers the next available number.
Thesaurus	Indicates that a multi-parent thesaurus should be created.
Obsolete	Obsolete records may be flagged rather then simply deleted.
Sum-Purge	Use summarize and purge techniques to remove old data.

Table 1: Commonly used Annotations

[10]. Query By Example, IBM's end user graphical query interface.

that effectively provide precedents and consequents that can describe any business rule, and these can be executed in the database itself. As most fields and blocks in 4GL programs are directly related to columns and tables in the database, ad hoc user interface annotations can also be placed on the conceptual schema. Modern 4GL structuring devices such as exception handling enable different code fragments to be sensibly combined.

Programs are harder to elegantly define than relational data structures so 4GLs contain more constructs than the dozen or so used by standard NIAM models. This has been reflected in the relatively greater number of annotations proposed in this paper. However, by concerning themselves mainly with the five general requirements listed in the introduction, and by using overrides and 4GL procedural annotations to cover unusual cases, a manageable number can significantly automate the production of realistic information system.

INTERMEDIATE SCHEMAS

Each form in the finished application can be described by a subschema that is extracted from the main schema as shown in figure 5 Experience with the Intelligent Developer showed that information systems could be designed in terms of three main interface paradigms, namely the selection, maintenance and reporting of Central entities (embolden on the figures, eg. Paper) together with selected facts that are related to that entity. Each fact in the subschema corresponds to a field in the form, with the paradigm defining how they are used in the interfaces.

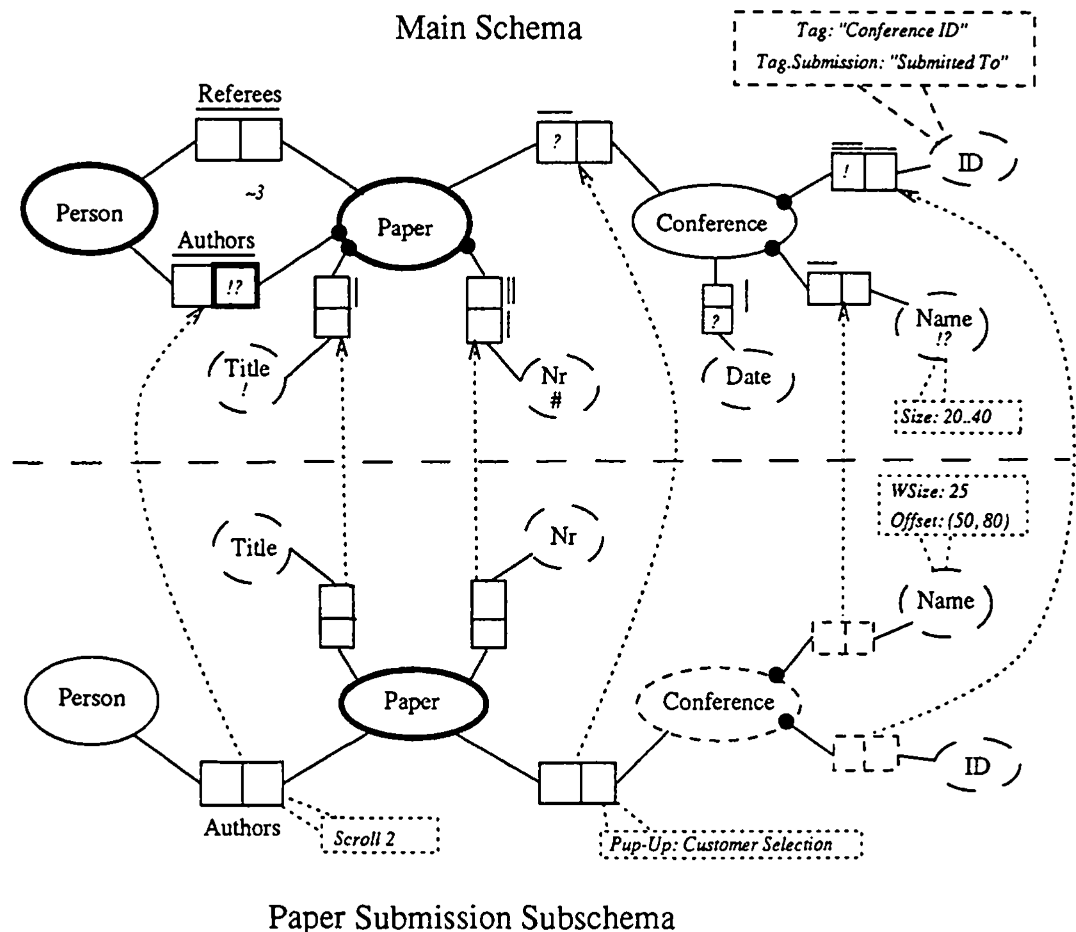

Figure 5: The Paper Submission Interface Subschema.

Unlike Pizano (1993), the subschemas are usually temporary data structures that are automatically extracted from the main schema during application generation, and destroyed once 4GL code has been produced. However it was found to be convenient to allow programmers to store and edit the

subschemas for the few interfaces that would otherwise have required excessive annotations on the main schema. The problem of maintaining stored subschemas as the main schema evolves is reduced by representing schemas by semantic nets, with each node in the subschema inheriting slot values from the corresponding node in the main schema, as shown in the figure. Thus stored subschemas only contains information that overrides information in the main schema so non-structural changes to the main schema will usually be automatically reflected in any stored subschemas.

Conceptual macros enable commonly used high level constructs to be defined in terms of annotated conceptual schemas. For example, the Obsolete flag on Subjects in figure 3 is used to flag subject categories that are no longer considered useful after librarians change the classification scheme. Obsolete subjects cannot be simply deleted from the database because existing papers that refer to them would need to be reclassified, which may not be considered worthwhile. Obsolescence is fairly common in practice, for example Institutions, Status codes, and Products may also be considered obsolete.

Figure 6 shows a conceptual macro that can automate the implementation of entities that can be flagged obsolete. It indicates that if an "&Entity&" such as Subject is marked *Obsolete*, then an Obsolete fact should be attached to it and the associated code executed whenever a reference to Entity is created. Different code may be generated for different entities, in this case "*&Long-Name(Entity)&*" will expand to the Long-Name meta-value of the primary entity. Generalized functions that could be implemented using conceptual macros include multi-parent thesauruses, mailing labels, summarizing and purging old data, and maintaining statistical histories.

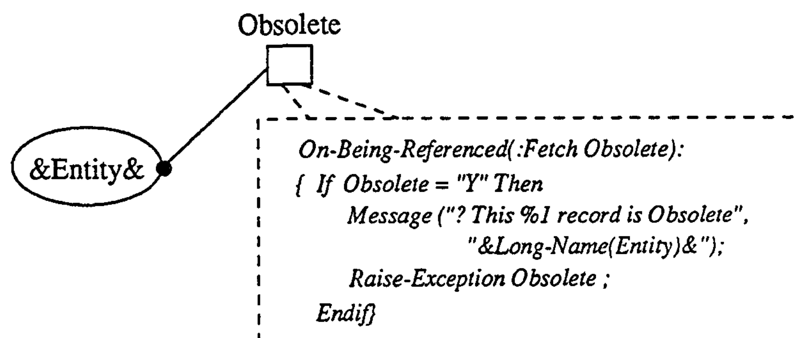

Figure 6: The Obsolete Module.

Conceptual macros are similar to super-classes in an object oriented system — saying that an entity can be *Obsolete* is analogous to saying that the entity inherits the Obsolete superclass. However, conceptual macros can access and manipulate the conceptual model quite generally — the Intelligent Developer uses the full power of Common Lisp as its macro language, which was found to be useful when developing realistic modules. Although implementing conceptual macros requires care, it is much easier than developing conventional macros because they augment conceptual models rather than generating 3GL code. For example, having attached the obsolete fact to the Subject entity, the obsolete module does not have to decide how to rearrange user interfaces to include the corresponding obsolete field. This has enabled application programmers using the Intelligent Developer to effectively extend the tool to suit their specific environments.

Normally only the invocation of the macro is stored in the main schema, and it is expanded each time it is needed. This makes it easy to change parameters to the macro, or even the macro's definition. However, programmers may elect to store and then edit a macro's expansion in the same way that they can edit stored subschemas.

USING THE ANNOTATED MODELS

Realistic information systems may involve hundreds of entities, so a major difficulty in developing them is to understand their overall structure. It seems unlikely that any single diagram like figure 3 could display all the details of a specification and at the same time be suitable for gaining an overview

of a large system. In order to resolve this conflict, CASE tools can display different abstractions of a specification, with changes made to one view automatically reflected in the others.

An initial design can be sketched out using an ERA view of the schema, and be partitioned into overlapping "scopes". The analyst can then expand a scope into an annotated NIAM diagram similar to figure 3 , but without the detailed annotations and expanded conceptual models shown in dashed boxes in that diagram. Those details can be displayed on demand when the analyst selects specific facts, entities or conceptual module invocations. New facts added to the NIAM view would have any corresponding relations automatically included on the ERA abstraction[11]. The analyst may also annotate certain ERA entities and relationships to be *Important*, in which case an overview diagram containing just those components can be displayed[12].

This style of interaction in which the analyst/programmer "zooms" in and out through different levels of abstraction is in sharp contrast to the one way journey through a waterfall model offered by conventional CASE tools.

End users may also wish to edit conceptual schemas. For example, different organizing committees may wish to store different fact types about papers that are submitted to a conference — one conference may be interested to know if a paper could qualify for a student paper award, while another might be interested in each paper's security clearance. In order to avoid having to modify a conference management system package for each individual customer, a trend in the design of information systems is to include generalized fact types such as Paper-has-Value-for-Attribute so that the end user can configure the names of any additional paper attributes in special code tables Simsion (1994). This fact type would produce a table that contains tuples like (<PAPER-123, SECURITY, X23>, <PAPER-456, STUDENT, TRUE>).

However, it can be inefficient to implement the "verticalized" tables that result from generalized data structures, the resultant user interfaces are often inelegant, application programs become more complex and the resultant application can still only be configured in ways that were conceived when the application was developed. As an application is made more generalized to enable users more configuration choices, the implementation problems increase. These issues are developed in detail in Berglas (1993a).

By enabling applications to be largely defined at the conceptual level, it becomes possible to enable end users to edit the conceptual models themselves and so avoid these generalized structures. Data entry operators may not have the skills to do this, but most organizations contain "spreadsheet literate" users that can build the simple spreadsheet applications upon which many important management decisions are based. These users could not design complex information systems or edit 4GL programs, but they could certainly learn how to add a simple fact to an existing schema and then have it automatically included in all the relevant interfaces.

In order to prevent spreadsheet literate users from corrupting the essential integrity of an application, special annotations may be used to specify how different end users may alter the conceptual models themselves. Adding new fact types is usually safe, as is changing user interface annotations on existing ones. End users cannot normally change the semantics of existing facts or entities unless the programmer specifically annotates them "Δ", although they may change any facts they created themselves and are therefor annotated "∇". Thus annotated models enable simple applications to be highly configurable without recourse to complex generalized designs.

CONCLUSION

By recognizing the fact that most information system functionality involves accessing, editing and disseminating data using common techniques, this paper shows that annotations can be used to augment conceptual models so that they provide a complete description of production quality applications without the need for advanced theorem proving technology. It also argues that a manageable number of annotations are sufficient and that a minimal amount of procedural code needs to be specified in practice.

Annotations provide additional parameters that make it feasible to develop sophisticated rules that can significantly automate implementation. Many of these rules are concerned with producing elegant user interfaces, the construction of which has been studied outside the information system context.

---

[11]. Ideally, the CASE tool should contain a schema layout algorithm.

[12]. Ie. *Important* annotations describes how the schema itself is used.

Annotated conceptual models are at the same time *concise* because they are used in conjunction with these rules and *complete* because impure annotations can override and augment any rule. Subschemas that describe individual interfaces can be explicitly extracted from the main schema, and may be stored and edited to avoid excessive annotations where necessary. It is also possible to produce libraries of conceptual macros that enable reuse at the conceptual level.
However, as far more resources are spent maintaining applications rather than in their initial development, a major benefit of using annotations is to enable analyst/programmers to "zoom" between different levels of abstraction rather than taking the one way journey through a waterfall model. Another important benefit is to enable end users to edit conceptual models in a well defined manner and so produce more flexible applications without recourse to complex generalized data structures.

ACKNOWLEDGMENT

Assistance in preparing this paper was gratefully received from Bob Colomb of the University of Queensland. The Intelligent Developer was written for and is marketed by BHA Computer of 1/303 Coronation Drive, Milton, Brisbane, Australia.

BIBLIOGRAPHY

Berglas, A. (1993a) "Enabling end users to tailor conceptual models in a disciplined manner", N. Sarda (ed.) **Conference on Information Systems and Management of Data. Insdoc.**

Berglas, A. (1993b) "The Intelligent Developer — towards the automated implementation of information systems" In M. Orlowska and M. Papazoglou (eds.) **Advances in Database Research, World Scientific.**

Berglas A. (1994) "Using KL-ONE knowledge representation techniques to structure active database rules" In R. Sacks-Davis (ed.) **Advances in Database Research.** World Scientific.

BHA Computer Ltd. (1992), Brisbane. **The Intelligent Developer Reference Manual**

Borras P., Mamou J., Plateau D., Poyet B., and Tallot D. (1992) "Building user interfaces for database applications: The O2 experience" **Sigmod Record**, Vol 21 No 1.

Brachman R. and Schmolze J. (1985) "An overview of the KL-ONE knowledge representation system" **Cognitive Science**, Vol 9 pp 171–216.

Flynn B. and Maier D. (1992) "Supporting display generation for complex database objects" **Sigmod Record**, Vol 21 No 1.

Halpin T. and Orlowska M. (1991) "Fact-oriented modeling for data analysis" **Journal of Information Systems**, Vol 2 No 2

IFIP (*Review of Information Systems Design* 1982) "The IFIP example" In W. Olle and H. Sol (eds.) **Comparative Methodologies**, pp 8–9. North-Holland.

Jackson M. (1984) **System Development**. Prentice-Hall.

Jarke M., Mylopoulos J., Schmidt J., and Vassiliou Y. (1992) "DAIDA: An environment for evolving information systems." **ACM Trans. on Information Systems**, Vol 10 No 1.

King R. and Novak M. (1993) Designing database interfaces with dbface. **IEEE Trans. Software Engineering**, Vol 11 No 2.

Oracle Corporation. (1993) **The CASE*Generator Reference Manual.**

Ovum Ltd (1992), London. *Ovum Evaluates CASE Products* Pizano A., Lizawa A., and Shiroto Y. (1992) "An automatic screen layout generator for database applications" In Y. Yesha (ed.) **Proceedings of the Conference on Information and Knowledge Management, CIKM-92,** pp 239–247.

Pizano A. (1993) "Automatic generation of graphical user interfaces for interactive database applications" In B. Bhargava, T. Finin and Y. Yesha (eds.) **Proceedings of the Conference on Information and Knowledge Management, CIKM-93.**

Reubenstein H. and Waters R. (1991) "The requirements apprentice: Automated assistance for requirements acquisition"**IEEE Trans. Software Engineering**, Vol 13 No 3.

Simsion G. (1994) "Implementation of very generalized data structures" In R. Sacks-Davis (ed.) **Advances in Database Research**. World Scientific.

Wiecha C. (1990) ITS: A tool for rapidly developing interactive applications. **ACM Trans. on Information Systems**, Vol 8 No 3.