Use of a Persistent Graph Abstract Data Type for Representing CASE Tool Repositories

Robert M. Colomb, PhD
Department of Computer Science
The University of Queensland
Queensland 4072 Australia
Email: colomb@cs.uq.edu.au

Abstract

This paper presents a persistent graph abstract data type which is suited to the implementation of repositories for CASE tools. It contains both deductive database and persistent graph representations of repositories for several CASE tools, in particular the entity-relationship-attribute method for conceptual data modelling. It shows that while the deductive database approach is feasible, a PGADT implementation would be superior.

Keywords persistence, graphs, abstract data types, SQL3, deductive database, CASE tools

1. Introduction

There are many applications requiring persistent storage of complex objects. One such application is a repository for a CASE tool. There are a large number of methods supporting various aspects of different classes of software development, nearly all of which are represented to the software engineer as a collection of symbols linked in a graph structure.

There are two main technologies used to store the structures used in such specifications: object-oriented databases, and relational databases (including related technologies such as logic programming). Object-oriented databases (Sagawa [27], Rine [25], De Antonellis et al. [9]) are perhaps more common. Object-oriented databases have the advantage that they offer persistent storage of the complex structures needed. In principle, object-oriented approaches also offer ease of implementation of reusable structures and methods. In practice, however, reusable object libraries are difficult to design (ICSE [17]), since to be easily useable they must provide an intellectually coherent and easily understood set of abstractions. The approach taken in this work is to develop a persistent abstract data type for graphs which, if implemented in object-oriented technology, could provide such a set of abstractions.

A disadvantage of object-oriented technology is that since all of the methods operating on the complex objects are provided by the user, the database manager is unable to provide a priori a rich set of tools for searching the data (Elmasri & Navathe [11]). The relational approach has the advantage that the CASE tool can make use of the rich set of functions available in a relational database manager (see for example Dampney & Colomb [7]). However, the implementation of complex objects in a relational system requires complex table structures (e.g. Markowitz & Shoshani [19], Mitschang [21]). In particular, as we shall see below, many of the tables have very complex reference schemes.

Our approach is to augment the relational database manager with a persistent store for graph structures, implemented as a relational application in a deductive database. This persistent store is presented to the programmer of a CASE tool repository as a set of views embodying a generic set of graph computations—in short as a Persistent Graph Abstract Data Type (PGADT). This ADT reduces the semantic distance between the CASE tool application and the implementation platform, simplifying the problem of implementing such repositories. The PGADT could also be implemented in an object-oriented database, but this has not so far been pursued.

To develop our argument, we first show examples of metamodels for the widely-used Entity-Relationship-Attribute (ERA) database conceptual modelling tool, including what sorts of functions a CASE tool expects to support their respective repositories. A relational implementation of a CASE tool repository is sketched. We then describe a PGADT, and show how this tool can be modelled in it. A discussion section compares the two representations, argues that the PGADT representation is simpler, and suggests further uses for the PGADT. Related work is then described.


1Work performed partly while visiting Department of Computer Science Heriot-Watt University, Edinburgh UK
2Work performed while visiting The University of Queensland.
2. Sample Metamodel for CASE Tools

We will now illustrate a fairly realistic repository which will support a substantial fraction of the ERA conceptual data modelling method, as described in Elmasri & Navathe [11].

There are three main constructs in ERA models: entities, attributes and relationships. The principal type of entity is a regular entity, since an instance is identified by an (possibly composite) attribute. There is another type of entity, called weak, an instance of which is partly identified through a special type of relationship (an identifying relationship).

A relationship is a regular relationship if it is not an identifying relationship for a weak entity. A relationship can relate any number of entities.

There are three types of attributes: key (underlined), descriptive, and derived (marked with *). An attribute can also be composite. Note that attributes can be associated with relationships as well as entities. Weak entities can have partially identifying attributes.

Attributes can be either mandatory or optional (dotted oval). An entity's participation in a relationship can be mandatory (thick line) or optional, and its participation can be either many or one.

Figure 1 is a conceptual model of the ERA constructs, which itself illustrates most of them. In particular, weak entities are rendered by double boxes, and identifying relationships by double diamonds. A relationship is identified by its name plus the names of all participating entities. (There are two relationships in Figure 1 named assoc with and two named identifies.) Note that the model uses the subtype construct, which to avoid undue complexity is not itself modelled.

Observe that the repository models the specification at a low level. The icons box, diamond and oval are all renderings of different meta-entity types. Their different renderings are the result of the values of attributes in the repository. (Note that the rendering of the partial identifier of the weak entity relationship is a result of treating participation in the relationship partial identifier as an attribute of the entity attribute.) The lines in the rendering derive from the relationships in the repository. Note that the relationship participates in has attributes, which determine whether the line between an entity and a relationship is rendered thick or thin (mand? true or false), and whether the line is annotated with N (M) or 1 (many? true or false).

Bernstein & Dayal [2] present a number of requirements for CASE tool repositories. Adapting these to the present example, the repository for the ERA method should enforce well-formedness conditions, such as
E1: Every relationship must have at least two participating entities;
E2: Every attribute must be associated with exactly one entity or many-to-many relationship;
E3: Every entity must be associated with the attribute which identifies it;
and permit a range of simple queries, such as
E4: Identify entities, relationships or attributes by name;
E5: Identify model elements by properties (e.g. all weak entities);
E6: Identify elements by structural patterns (e.g. relationships with attributes, relationships of degree greater than 2).

Complexity of structure arises from
E7: Need to propagate updates.
E8: Module structure needed to support check-in/check-out for workflow control in updates, and also version control.

These queries can be represented using SQL. (E2 requires a recursive union, since attributes may be composite, as does E7, so SQL3 is required.)

An ERA model has also a large-scale structure. In particular, an important notion is that of functional dependency. An entity on the one side of a binary relationship is functionally dependent on entity on the many side. Below (1) is partial relational implementation of the model in Figure 1. Note that only binary relationships and elementary attributes attached to entities can be stored in the tables shown.

```
Entity(Name, Ident-Att, Weak?) (1)
Relationship-2(Name, E Name1, E Name2, Many?, Many?)
Participates(Ent Name, Rel Name, E Namel, E Name2, Many?, Mand?)
Entity-Attribute(Att Name, Ent Name, Comp?, Deriv?, Opt?)
```

We can define a view which will return functional dependencies among entities, whose domain is a single entity, which will help support E7. Functional dependencies are transitive, so that a recursive view definition is needed.

```
CREATE VIEW EFD (DomEnt, RangeEnt) (2)
SELECT PD.Name, PR.Name
FROM Participates PD,
Participates PR
WHERE
PD.Rel_Name = PR.Rel_Name
AND
PD.E_Name1 = PR.E_Name1
AND
PD.E_Name2 = PR.E_Name2
AND
PD.Many? = True
AND PR.Many? = False
RECURSIVE UNION
SELECT EFD.DomEnt, PR.Name
FROM EFD, Participates PD,
Participates PR
WHERE PD.Name = EFD.RangeEnt
AND
PD.Rel_Name = PR.Rel_Name
AND
PD.E_Name1 = PR.E_Name1
AND
PD.E_Name2 = PR.E_Name2
AND
PD.Many? = True
AND PR.Many? = False
```

Finally, CASE tools often must record properties of large scale structures in their specifications.

For example, in Figure 2 (adapted from Dampney et al. [8]) is shown two indirect functional dependencies between the entity Picking and the entity Product. An instance of Picking is related to an instance of Product once through Order Item and once through Bin. The business requires that the two instances of Product be the same (the product picked from the bin is the product specified in the order item). The functional dependencies specified by the relationships R1 - R4 are not sufficient to insure this, so that an additional
constraint must be specified by the analyst. The situation of two different functional dependencies between two entities is called a diagram, and the constraint is that the diagram is said to commute. (Some diagrams in a conceptual model must commute and some may not. See Dampney et al. [8] for further details.)

To record the commuting diagram constraint, the repository must be able to identify these structures, and must be able to support integrity constraints such that if a diagram is destroyed by removal of one of its functional dependencies the constraint can be removed. Contrariwise, if a new functional dependency is entered, new diagrams may arise, and the analyst must be alerted to the possibility that the new diagrams commute.

One way to identify a diagram is by its two paths, so that the repository might have a table

\[
\text{Diagram(Path1, Path2, Commutes?)}
\]

Since the diagram is the same whichever order the paths are considered, we would impose the additional constraint that

\[
\text{Path1} < \text{Path2}
\]

which requires that the path identifiers be capable of supporting an order relation.

A natural way to construct path identifiers is to modify the indirect functional dependency view (2), adding a constructor function \(\text{Path_ID} / 3\) for path names and a constructor function \(\text{Path_ID_EXT} / 3\) for extending path identifiers. We need a view to identify diagrams, building on the functional dependency view EFD (2). To support the integrity constraint we might specify insertion and deletion triggers on this view.

We have thus far argued that a relational implementation of a repository supporting the ERA conceptual modelling method is plausible, given some plausible extensions to relational technology:

- the ability to compute linear recursive queries such as (2), which is included in the proposed SQL3 standard (Melton [20]);
- the ability to compute composite identifiers for paths which is included in many experimental deductive databases, e.g. Aditi (Vaghani et al. [31]);
- the ability to support triggers on views rather than base tables, which is a feature of many experimental active databases, as surveyed e.g. Widom [32].

The relationship between relationship identifiers and path identifiers is probably too complex to be supported directly by the automatic mechanisms suggested to date in deductive or active databases, but could be supported by making use of intermediate representations, such as a table associating path identifiers with relationship identifiers.

3. Beyond Deductive Database

At this point, the reader should be convinced that it is, or at least is about to be, practicable to implement CASE tool repositories using deductive databases, and should have an idea of what they might look like.

We can now turn to some software engineering issues. First, we consider simplicity of representation. The repositories we have illustrated are fairly small as data models go, with only a few entities and relationships. However, these entities are individually quite complex- weak entities require a large number of identifiers in their relational implementation. In addition, the SQL language used requires a fairly complex specification of what are fairly simple ideas in the graphical realization of the structures held in the repository. The commuting diagram of Figure 2 is one example. We examine here a more complex example.

![Figure 3: Elimination of Many-to-Many Relationships](image)

A consideration of an ERA model in terms of functional dependencies can require that the model be normalized, in that many-to-many relationships are replaced by a new (weak) entity and (identifying) many-to-one relationships, as shown in Figure 3. The relationship R is replaced by the weak entity ER. The M side of R is replaced by the identifying relationship R1, and the N side by the identifying relationship R2.
The many sides of relationships R1 and R2 are mandatory, while the one sides are mandatory only if the corresponding sides of R are. (Normalization also requires attachment of relationship attributes to the new entity, and further, the replacement of ternary and above relationships with weak entities.)

This normalization can be accomplished with view definitions expressed as graph operations. First, the new entity ER is formed from the old relationship R. We then need two new relationships, one replacing the arc from R to E1 and the other replacing the arc from R to E2. Finally, the arcs from R1 and R2 to ER must be mandatory and many.

In the relational implementation of this normalization, we need six view definitions, each involving several tables with complex identifiers. The graph representation is far simpler.

Green [13] describes a number of what he calls cognitive dimensions, which are used in the design of representations through which humans can understand and operate with complex structures such as are encountered in the human-computer interface.

Two dimensions are especially relevant: diffuseness (number of symbols per idea) and presence of perceptual cues. The representations we have used score poorly on the diffuseness dimension: a large number of symbols are required to represent most of the ideas in the earlier part of the paper, even though the ideas themselves are fairly simple. We have already mentioned that the algebraic representation of the structures and operations loses the perceptual cues present in the graphical representations.

In fact, much of the motivation for the use of CASE tools in software development is to reduce diffuseness and to increase perceptual cues. This forms part of the explanation for the use of object-oriented technology as discussed in the Introduction: the data structures and operations employed are designed to be close to the application.

We now consider a pair of software engineering principles: cost of construction and reliability of product. The great strength of the relational data model is that the database managers which implement that model permit relatively untrained people to produce a good quality product at a low cost: the database manager provides a rich set of high level operators so that systems involving a small number of entities and relationships can be implemented with very few statements. Pure object-oriented databases provide very little such support. (These issues are discussed in Elmasri & Navathe [11]). Although the programmer of a repository in an object-oriented database can produce a reliable system which is structurally close to the application, it takes a great deal of effort to do so.

Our proposal is to retain the relational model, but to recognize that much of the structural complexity of the repository arises from the fact that the models represented are graphs, and that many of their relevant properties are graph properties. We therefore augment the deductive database with structures and operations from the theory of graphs, and express the repository largely in this new graph abstract data type. This approach factors out much of the complexity seen above: in particular the need for the programmer to construct recursive views is greatly reduced, if not eliminated. (The recursive views themselves are not eliminated, they are provided pre-built by the database manager, in the same way as the complex procedures needed to compute joins.)

In the next section, we describe the abstract data type and illustrate the implementation of the ERA repository. Following is a discussion, then a section on related work.

4. Persistent Graph Abstract Data Type
A graph ADT will have a number of components:

- Specification of its elements: nodes and arcs;
- Ability to identify complex constructs, such as paths;
- Complex graph-theoretic functions;
- A query language;
- Update capability, with integrity constraints specified using the query language;
- Integration with relational operations.

4.1 Elements
A graph consists of a set of nodes and a set of arcs. Each node must have an identifier, and in most applications, nodes have properties. In the ERA application in Figure 1, all of the constructs entity, attribute and relationship are modelled as nodes, and each of them has additional properties.

Each arc is associated with two nodes. If we model directed graphs, then one of the nodes is the source and the other is the target. It is consistent with the use of the graph in our application that the identifier of an arc include both the node identifiers (none of the arcs in Figure 1 are labelled individually). As with nodes, the application requires arcs to have properties.

It is convenient to separate out one of the properties to be a type. To store the other properties we need to be able to store a different number of properties for different types of object. There are several ways to accomplish this, but we will use a property-list-valued attribute for expository purposes, which consists of a set of attribute/ value pairs.

We therefore arrive at the following representations of the nodes and arcs as relations:

\[ \text{Node(Type, Identifier, Properties)} \]  
\[ \text{Arc(Type, Name, Source, Target, Properties)} \]

A graph instance therefore consists of two relation instances. We can adopt the dot notation to identify the arcs and nodes of a graph instance, by nodes =
This model should be compared with (I). Note that the repository is therefore modelled of implementing this, but for expository purposes we will employ Prolog-style structures. The ERA repository is therefore modelled

\[
\text{Node(entity, Identifier, \{weak? = W, ident_att = A\})}
\]

\[
\text{Node(relationship, rel_id(N, Name, \{E_Name1, E_Name2, \ldots\}), \{ident? = I, arity = A\})}
\]

\[
\text{Node(attribute, att_id(N, Attached_To), \{comp? = C, derived? = D, optional = O\})}
\]

\[
\text{Arc(a, n, Source, Target, \{many? = M, mandatory? = D\})}
\]

This model should be compared with (1). Note that the model (7) includes relationships of any arity, attributes attached to relationships, and composite attributes. Arcs have only one type, and are identified by their source and target nodes only.

Note that identification of subgraphs needed to support E8 is quite natural.

4.2 Complex constructs

The principal complex construct in graphs is the path. We have in the discussion of commuting diagrams above specified two constructor functions: \(Path_{ID/3}\) for path names, and \(Path_{ID\_Ext/3}\) for extending path names. These path name constructors could be implemented using list constructors.

There is an advantage in the ADT for having a specific path constructor over using a generic list facility, in that the ADT can in principle keep track of the relationship between path identifiers and arc/node identifiers when the persistent graph store is updated. This facility is important, since in many applications paths have properties, and we have seen above that it is difficult to maintain in general referential integrity between updates in arcs/nodes and complex structures containing them. The properties may be derived as aggregates from the properties of the included nodes and arcs, but evaluated eagerly and stored, or may be entered by the user, such as the 'commuting diagram' property discussed above (Figure 2).

A facility to construct generic lists or Prolog structures is also useful in many applications.

4.3 Graph-theoretic functions

The main benefit of a graph ADT is the provision of graph-theoretic functions, which can be implemented efficiently in terms of the graph elements, and which can be employed in applications.

Perhaps the most significant of these is the computation of a path, which is required for many CASE applications. In the ERA model, we have needed it for functional dependencies and commuting diagrams.

Since a path is the transitive closure of arcs, it makes sense for the path function to have the same signature as the arc (6), viz. \(Type, Identifier, Source, Target, Properties\). We have already considered the representation of the identifier of a path, and concluded that a list-like structure is required. By analogy, both the \(Type\) and \(Properties\) attributes should also be lists (respectively of the types and properties of the arcs in the path, in the same sequence as the identifiers of the arcs in the path identifier). There is also an argument for \(Type\) to be a simple attribute, and that a path be restricted to arcs of the same type.

A complication is that the CASE structures we are considering sometimes have graph properties at several levels of granularity. Consider in particular the functional dependency in the ERA model (2), where the dependency is represented by two arcs (from a relationship node to the source entity node, and from the relationship node to the target entity node). It would therefore be an advantage if the path computation function were parameteric with respect to the definition of an arc, which would be any function with the same signature as the arc.

The signature of \(p\) path would therefore be

\[
\text{Path}(\text{Type}, \text{Identifier}, \text{Source}, \text{Target}, \text{Properties}, \text{Arc})
\]

where \(Arc\) is the name of a view with the signature (6). (The actual definition of these views depends on the query language supplied, which will be considered below.)

There are other functions closely related to \(p\), including \(\text{cycle}\) (path where the source and target nodes are the same) and \(\text{diagram}\) (pair of paths with the same sources and targets, but different path identifiers).

There are also more complex functions which can be useful in CASE applications. An obvious such function is \(d\) dual, which reverses the direction of the arcs. \(Dual\) has the signature

\[
dual : \text{arc definition} \to \text{arc definition}
\]

Conceptually, dual/1 allows one to construct new view definitions through which the arcs seen in the input arc definition view are reversed. Note that capabilities of this sort require implementation as parameterized functions with, at the database level, relations as values. Implementation of the PGADT would be simplified if the underlying database manager supported this.

A useful graph-valued function is connected component, which computes a connected component rooted at a nominated node. In the ERA model, if the connected component were parameterized as is \(p\), then the set of entities functionally dependent on a
nominated entity is precisely the nodes in a connected component rooted at the nominated entity. The signature of connected component is therefore

\[
\text{conn_comp : node } x \\
\text{arc_definition : } x \text{ graph } \rightarrow \text{ graph }
\]  

The two functions dual and connected component interact. For example, in the ERA model, the connected component of the dual of the graph rooted at a given node is the set of entities which, if changed, would affect a nominated entity. This provided a simple specification of E7.

There are many such functions, well-defined in graph theory, which might be useful in some applications.

4.4 Query language

The graph ADT is built on top of a relational database, with the nodes and arcs specified as relations. It therefore makes sense to employ one or more of the well-developed database query languages such as SQL, QBE, or the user interface systems built upon them. These sorts of languages can be employed to select subgraphs based on properties or to find individual nodes or arcs.

Use of such database query languages for a graph abstract data type is complicated somewhat by the representation of a single complex object (a single graph) by two relations (nodes and arcs). Selection on the nodes relation may result in arcs with no source or target node in the selected set, and selection on the arcs relation may result in isolated nodes. The latter situation is a well-formed graph, but the former is not.

Graph-valued queries must therefore consist of two queries, one for the nodes and the other for the arcs. It would be convenient for the query language to allow the result to be named, ideally with a default name such as RESULT. By default, the arcs component of a query on the nodes would be created automatically. For example, if one wanted the subgraph consisting of nodes of type 'T', the following query would be convenient

\[
\text{SELECT * FROM Graph.Nodes} \\
\text{WHERE TYPE = 'T'}
\]  

whose result would be accessed by default as RESULT.Nodes. The system would automatically generate the query

\[
\text{RESULT.Arcs =} \\
\text{SELECT * FROM Graph.Arcs} \\
\text{WHERE} \\
\text{From = RESULT.Nodes.Name} \\
\text{OR To = RESULT.Nodes.Name}
\]  

Furthermore, there have been a number of proposals for database query languages based on graph models (Poulovassilis & Levine [24], Consens & Mendelzon [5], Gysseens et al. [14], Paredaens et al. [22]). Of these, the Graphlog language (Consens & Mendelzon [5]) is particularly relevant to the present work.

Graphlog is equivalent to linear stratified datalog, and can therefore express queries involving transitive closure. Queries are expressed as graph patterns, and responses represented as arcs. The queries involved in normalization of an ERA model can be expressed, in a form something like Figure 2. It is likely that the language could be adapted to actually specifying the transformation indicated in Figure 2.

Since Graphlog specifies queries in terms of graph patterns, there is considerable scope for graph-based visualizations of queries and results. A possible approach is to make use of one of the graph editing systems which have been developed in recent years (Frohlich & Werner [12], Himholt [15, 16], Paulisch & Tichy [23]).

The query language needs also to have integrated into it the graph functions described in the previous section, and probably also a facility to define and integrate user-specified functions of similar complexity.

4.5 Update capability

Any persistent data type needs the ability to update the persistent objects. It is also very advantageous if the persistent store manager can automatically maintain a set of integrity constraints- this simplifies the programming tasks considerably.

Since the graph ADT is built on a relational database, update capability is already present, as is the ability to specify some kinds of integrity constraints. Integrity constraints used to maintain consistency in the graph structure of an application may be more easily expressed in a language such as Graphlog.

Examples of integrity constraints needed to maintain consistency in an ERA model include:

- an elementary attribute node is the target of an arc.
- the arcs involving composite attributes are directed from the entity or relationship out.
- an entity is the target of an arc involving a relationship.

These could all be easily expressed as graph patterns.

4.6 Interaction between graph functions and relational operations

An advantage to development of the PGADT in a relational database environment is that relational operations can be combined with graph operations to give a simple representation to concepts which are fairly simple in the application domain but are complex in either the database domain or the graph domain. For a plausible example, we need a more complex CASE environment than so far presented. Dampney & Colomb [7] show a relationship between a process specification system and the ERA data modelling system. In it, the process specification consists of a set of interconnected named Finite State
Machine (FSM) models. The state vectors for the FSM models must be represented in the ERA model. The authors propose the constraint that the state vector for each FSM be represented in the ERA model as attributes which are functionally dependent on a single entity, called the central entity for that FSM.

For the purpose of the present example, we do not require much detail from the process model repository. Assume a table containing the name of a FSM and the date on which it was last modified:

\[
\text{PROCESS(State Machine, Date Modified)} \tag{13}
\]

and assume that the relationship between the two models is held in a table LINK_PE with the schema

\[
\text{LINK_PE(State Machine, Central Entity)} \tag{14}
\]

Suppose the analyst responsible for a particular entity \( E \) in the ERA model wishes to know whether any changes made in the process model since a given date \( D \) have the potential to affect entity \( E \). We have noted above that the entities whose change would potentially affect a given entity is the connected component of the dual of the functional dependency graph rooted at the entity in question. Assume that the functional dependency connection is given by the view FD. The desired query would be a join (15), using definitions of dual (9) and connected component (10)

\[
\text{SELECT * FROM PROCESS, LINK_PE, conn_comp(E, dual(FD), ERA MODEL) C}
\]

WHERE

\[
\text{PROCESS.Date Modified} \geq D \text{ AND LINK_PE.State Machine = PROCESS.State Machine AND LINK_PE.Central Entity = C.Nodes.Identifier}
\]

Such a definition could be further simplified if the PGADT allowed the user to specify domain-specific aliases for functions. The function in (15) could be expressed as (16)

\[
\text{entities_affecting_entity}(X, Y, Z) = \text{conn_comp}(X, \text{dual}(Y, Z), Y) \tag{16}
\]

5. Discussion

We have developed a sketch of a specification for a repository for the ERA data modelling tool, in an SQL3 deductive database environment. We have then defined a persistent graph abstract data type and re-expressed the ERA tool in it. It should be apparent that the PGADT representation is simpler: consider in particular the maintenance of the commuting diagrams view, and the relationship between two models given in (15). A PGADT equipped with the Graphlog query language, supported by a graph editor, would be a powerful tool, indeed.

5.1 Implementation considerations

We have described a specification for a generic persistent data manager. Its practicality depends greatly on whether its implementation provides sufficient performance for its applications. The pilot implementation of the core of the work was implemented in SICStus Prolog (Carlsson [3]). The examples presented in the work using SQL3 have been produced by hand translation from the Prolog originals. This implementation gave adequate performance for the small examples of ERA (and also FSM, not reported here for lack of space) specifications used as test cases.

In general, the performance of such a system depends greatly on its underlying platform, while the adequacy depends on salient characteristics of the application. The PGADT has been specified as an extension to a (deductive) relational database. Relational databases have been implemented on a wide variety of platforms, including main-memory (Lehman & Carey [18]). One would expect that SQL3 implementations will follow a similar pattern. The graph-theoretic extensions can be implemented in the database programming language provided with the SQL-3 platform. A main-memory database implementation would be expected to give good performance for applications where the databases are not extremely large (in which the data exceeds what can be held in main-memory), and where the complexity of the graphs searched is moderate (with very complex graphs, execution of the generic graph algorithms will tend to dominate the resource consumption).

The proposed application, as a repository for specifications developed with CASE tools, fits these dimensions well. A very large ERA model would have 1000 entities and relationships, while the maximum length of a chain of functional dependencies would rarely be more than say 4.

5.2 Other applications

First, there are other classes of application which have similar dimension to CASE tools. These include other design systems, natural language processing and artificial intelligence systems using semantic nets or conceptual graphs as data structures, and planning applications. Any application for which an object-oriented database is appropriate could be considered for implementation in the PGADT.

Second, the PGADT could be the basis for interchange of complex structures. The existing relational data model supports interchange of data (via SQL DDL statements) and potentially complex views (via SQL DML statements). Exchange of object methods requires exchange of procedural code (CORBA [6]). Since the functions supplied with the PGADT are widely agreed upon, exchange of specifications of complex processes could be done declaratively, using a
set of standard definitions and signatures based upon the mathematics of graphs.

Finally, and more speculatively, the PGADT could potentially be used as a platform upon which to build structures using more advanced mathematics, in particular category theory. This mathematics is used in many parts of computer science, including information systems (e.g., Dampney et al. [8]). Rydeheard and Burstall [26] show that the constructs and operations of category theory can be implemented in a programming language, which gives some plausibility to the notion.

6. Related Work

Abstract data types for CASE tools are often built using an object-oriented approach (Rine [25], De Antonellis et al. [9]). In order to access the functions of a class, other classes must inherit access to its methods. Therefore, the classes which are to have access to certain functions must be decided in advance. When extending the classes, a new operation must be carefully placed in the correct class, with all necessary access to the required functions of the previous classes.

In our work, since the ADT is constructed bottom-up, all the basic functions are available to any procedure using the ADT.

Some repositories do not employ object-oriented technology. For example, in Casanova et al. [4], an ERA repository is constructed using a specially designed implementation of Prolog. The model stored in the repository can be translated into database schemas. The ADT in that work consists not only of operations for retrieving information from the ERA schema, but also the transformation rules from the model to the relational implementation. The rules in the ADT are highly specialized to the ERA problem, while in our work, the focus is on abstracting functions common to many problems.

Earlier work (Sagawa [27]), upon which the IBM AD-Cycle product suite was based, represented the CASE tool in two parts: data and functions. The data part is responsible for the presentation and integrity policies, while the function part is responsible for the functions and derivation rules. One can say that the function part is a sort of ADT which operates on the information in the data part. Other parts of the tool can make use of the information in the data part, and can be viewed as an ADT for it. Again, these ADTs are too specific for our purposes.

Several researchers have used deductive database-style repositories for CASE tools. The work of Casanova et al. [4] has already been cited. Structure chart and data flow diagram models have been built in Prolog (Tse et al. [28]).

There has been work in developing graph abstract data types. Beierle and Pletat [1] focus on feature graphs, taking a theoretical approach. Atoms are nodes, features are arcs, and the main focus is on arcs, which are directed. The graph has a distinguished root node. There are syntactic and semantic parts to the ADT, including consistency, completion and unification. The focus is on connectivity properties, so that the model lacks the ability to attach attributes to nodes and arcs. It is also not implemented.

Ebert and Franzke [10] use graph-theoretical concepts to describe complex integrity constraints. The graph structure is described using a declarative specification language GRAL, which permits nodes and directed arcs with types and attributes. The ADT operations are based on graph theoretical operations. The graph and ADT are expressed in a specialized version of Z, which is converted to C.

GRAL permits specifications of the same sort of graph structures as this work. The operations, however, are focussed on finding paths in the graph. The GRAL specifications are specific to the application, and must be compiled into C code before execution, unlike the essentially interpreted database approach taken in our work. The embedding of the application into the graph structure makes representation of more than one application more complicated, and requires duplication of much basic code.

Poulovassilis and Levine [24] report a data model for graphs given by the pair nodes x edges (N, E) where node N can be itself a graph. The model is intended for implementation of hypertext databases. It is queried using language Hyperlog, which is a path expression containing positive and negative predicates on properties of nodes and edges. The graph is a data structure only. No graph-theoretic operators are provided. In this respect, Hyperlog is similar to Graphlog (Consens & Mendelzon [5]), GOOD (Gyssens et al. [14]) and G-Log (Paredaens et al. [22]).

Finally, this work is related to graph editors, since the CASE specifications are represented to the user as graphs. Graph editors are designed to represent nodes and arcs, and to automatically render graph structures on displays. Most can display different types of nodes and arcs, which makes them suitable for rendering the specifications in the repositories of the various CASE tools (Frohlich & Werner [12], Himholt [15, 16]).

These various programs are more or less capable of rendering the graph visualizations of various CASE tool specifications. They are generally seen as visualization tools implemented as libraries in C or C++. They lack good persistent storage capabilities, standard graph processing functions and the query and manipulation facilities offered in a database environment.

References


[8] Dampney, C.N.G., M. Johnson, P. Dazeley & V. Reich, "A higher-order commuting loop structure that supports very large information system data and process architecture" Proceedings IFIP TC8 Working Conference, May, Bond University, Gold Coast, Australia (1994).


