Applying Database Dependency Theory to Software Engineering

Darrell R. Raymond
Frank Wm. Tompa

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

ABSTRACT

We describe the use of database dependency theory for investigating software designs. Dependency theory captures some of the essential constraints implicit in a system, and focuses attention on its update properties. The fundamental choice between redundancy and normalization is directly related to the issue of reuse. We show how dependency theory can be applied to the design of text editors and spreadsheet systems, and discuss its implications for object-oriented programming.

December 31, 1992

Applying Database Dependency Theory to Software Engineering

Darrell R. Raymond
Frank Wm. Tompa

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

1. Introduction.

Database and software engineers share an interest in conceptual modelling. In software design, modelling evolved from the early use of data typing to control program errors. It has long been known that classifying or ‘‘typing’’ program objects enables the automatic detection of invalid or suspicious assignments, comparisons, and other operations. Data types also serve to make semantics explicit in a program. Instead of manipulating processor entities such as bytes and characters, programs can be written in terms of data types that are closer to the entities of interest. Modelling for software assumes great importance in the object-oriented approach, which is largely based on definition and classification of objects and their operations.

Conceptual modelling is also a basic task in database design. Modelling for databases was originally adopted as a means of making database systems less dependent on machine architectures[11]. Database engineers learned to build systems that were insensitive to machine or operating system features by specifying the design in terms of the constraints and dependencies of an abstract model. A conceptual model also reduces hidden dependencies on a particular (transient) set of data, and is a good interface for general-purpose query languages. Many data models have been developed[16], of which the best known are the relational and entity-relationship models. As in programming languages, object-oriented approaches are currently an important area for study by database practitioners.

Conceptual models serve several useful purposes. The main function of a conceptual model is to act as an abstract organizing concept that addresses the human problem of grasping complex systems. Conceptual models also encourage a separation of concerns; their abstract nature means they tend to separate implementation-dependent and time-invariant properties of a system. They thus induce a basic modularity in the design. Finally, many conceptual models are formally defined and efficiently implementable.

Abstract type theory would be much less interesting if it was not possible to efficiently test a program for type satisfaction. The combination of organizing power, separation of concerns, and formal tests for conformance is what makes a useful conceptual model.

Despite their common interest in modelling, database and software practitioners do not often strive for generality in their models[6, 17]. This separation has occurred despite the fact that many of the models are quite similar except for terminology and some application-specific issues. Thus, a unification of the two areas is in order. Our present attempt at unification is the extension of database dependency theory to address concerns in software design. This theory was developed to manage the update properties of relational databases, and is little used outside of database applications; we know of only one example[14]. Yet the update properties of non-database systems are clearly an important design issue. By attempting to extend dependency theory to these systems, we hope to explore the use of existing predictions of the theory, and perhaps discovering inadequacies in its power and range.

2. A brief overview of dependency theory.

Dependency theory grew out of the need to provide the relational database model with more semantics. Consider the following relational schema:

part	*supplier*	*quantity*	*location*

This relation holds information about parts, their suppliers, and the quantity of parts kept in certain warehouses or locations. If every row in the table is unique, then the table is an instance of a relational database[11].

While this table captures facts about a particular instance of parts, suppliers, quantities, and locations, it does not adequately express some potential relationships. We might require, for example, that every part come from exactly one supplier and that every location for a part has exactly one quantity of that part. As a result, the pair *part*, *location* is a key for the table; every instance of this pair has unique values for *quantity* and *supplier*. This information is implicit in the data values, but can be captured explicitly in the schema if we add the following dependencies:

$$\mathit{part} \rightarrow \mathit{supplier}$$
$$\mathit{part}, \mathit{location} \rightarrow \mathit{quantity}$$

A dependency is a relationship that holds over a set of possible instances. The two dependencies given above are *functional dependencies*: there is only one value of the right hand side of the dependency for a given value of the left hand side, known as the *determinant*. By binding these two dependencies to the description of the table, we ensure that there is only one supplier for every part and one quantity for every part-location pair, no matter what the values in the table.

Dependencies come into play when we try to modify the data: valid updates must leave the table in a state that satisfies the dependencies. If a potential update results in a violation of a dependency, the table is said to suffer from an *update anomaly*[11]. The parts database exhibits several types of update anomaly. One example occurs when adding information: if we want to add information about a part and its quantity, we must also know the supplier and the location of the part, because an addition must be a complete row.† A second update anomaly occurs when deleting information: if we delete the last part supplied by a particular supplier, then we will lose that supplier's name from the database. A third type of update anomaly is redundancy: if we have parts at many locations, the supplier name is repeated for each one. This introduces the possibility of inconsistency in supplier names, since it is possible to update one instance of the name without updating the others. Redundancy also leads to poor storage utilization.

Eliminating update anomalies makes the database more robust to updates. The process of removing anomalies is called *normalization*. Normalization generally involves decomposing the schema into two or more schemas. A normalized database stores the same data and obeys the same dependencies as does an unnormalized one, but it avoids some of the update anomalies. We can decompose the parts database into the following two schemas:

part	*supplier*

part→*supplier*

part	*location*	*quantity*

part, *location*→*quantity*

The revised schema has different update characteristics. It is now possible to maintain information about parts and suppliers without actually having the parts on hand; that is, we do not need a location or quantity. The redundancy in supplier names has also been eliminated, since the supplier needs to be mentioned only once for each part (and not repeated for each distinct location).

The ability to redefine schemas to remove update anomalies leads to the concept of *normal forms*. A normal form is a class of schemas that is free from a specific type of update anomaly. Database practitioners have identified several normal forms corresponding to freedom from different types of update anomaly. The original parts database schema, for

† It is possible, but not usually desirable, to add a row that includes null values for unknown information. This complicates the semantics and implementation of updates and other aspects of the system.

example, is not in *second normal form*, usually abbreviated 2NF. A schema is in 2NF if every attribute is fully dependent on the key. In the first schema, the key is *part, location*; however, *supplier* is dependent only on *part* and not on *location*. Thus, there exists an attribute that is not dependent on the full key. Normalizing the schema has put it into 2NF, and thus removed some of the update anomalies.

In summary, then, dependency theory adds semantics to database structures by expressing class-wide constraints. Normalization is the process of removing update anomalies, while still maintaining the same data and dependencies. A schema is said to be in a normal form if it is free of a specific class of update anomalies.

This description of dependency theory has been necessarily brief and informal. Readers who want more information on the subject are advised to consult the database literature[4, 13, 19, 30].

3. Text editors.

Most software systems are not database systems, but many of them manage information that is frequently updated. Text editors, for example, are tools whose main function is to update a text. The importance of update to editors is implicit in the vast research literature on syntax and structure-driven editors, which provide various kinds of tools for more advanced update of documents[18, 21, 27, 29]. Even the basic text editor design, however, illustrates dependencies and update properties of importance.

Consider the simple task of changing all occurrences of a single word in a document, for example changing ‘‘color’’ to ‘‘colour’’. In order to treat this task with dependency analysis, we must construct the schema that is embedded in the editor design, and then consider its update characteristics. A schema (and partial instance) for the editor task can be represented as follows:

position	*id*	*spelling*
124	23452	"color"
.	.	.
.	.	.
.	.	.
13	23452	"color"
.	.	.
.	.	.
212	23452	"color"
167	23452	"color"

$$\mathit{id} \rightarrow \mathit{spelling}$$
$$\mathit{position} \rightarrow \mathit{id}$$

This schema captures only the important conceptual structure of the editor, and not its actual implementation. Each row of the table indicates a *position* in the text, a *spelling*, and an *id* that denotes the word in question. Thus, in this document, the word with *id*

‘‘23452’’ is spelled ‘‘color’’. The *id* ‘‘23452’’ is simply an arbitrary number that is meant to uniquely identify the concept of the word ‘‘color’’. The functional dependencies indicate that each *id* has at most one *spelling*, and that each *position* in the text has at most one *id*. The schema is highly redundant; each *id* is represented once for every time it appears in the document. This is true of virtually all editors, which store a copy of each instance of a word. On the other hand, in most editors the *id* is present only in the mind of the user; that is, there is no explicit representation of the entity ‘‘23452’’, there is only the user’s perception that some set of identical strings constitutes a certain word.

The update properties of the editor can be deduced from the schema. To update the structure so that *id* ‘‘23452’’ is spelled ‘‘colour’’, we must change every instance of the spelling. This schema, then, exhibits the update anomaly of redundancy. This anomaly is well known to novice users of text editors, who perform the update by laboriously locating every instance of ‘‘color’’ in the document and changing the spelling by hand.

Now consider the schema from the viewpoint of a database engineer. The functional dependencies for the schema are *id* -> *spelling* and *position* -> *id*, and the schema’s key is *position*. The dependencies show that the spelling of a word is determined not only by the word itself, but indirectly by its position in the text. This is a corollary of the observation that we must go to every location of the word in the text to change its spelling. The situation is analogous to the parts database anomaly, where we needed to change the supplier name in several places. For the database engineer, the problem with the schema is that it is not in *Boyce-Codd normal form*. Boyce-Codd normal form states that for all functional dependencies, the determinant must be the key.† The intent of Boyce-Codd normal form, roughly, is to ensure that all the attributes are determined directly by the entity that they represent, and by nothing less. Since the key identifies the object, ensuring that every determinant is a key will ensure that the object itself determines the rest of the attributes. The key in this example is *position*. Since *spelling* is functionally dependent on *id*, and *id* is not a key (i.e., not a unique identifier for a row in the table) the schema is not in Boyce-Codd normal form.

We can decompose the schema to produce Boyce-Codd normal form. The result is the following pair of schemas:

† More accurately, that for every non-trivial functional dependency, the determinant must be a superkey. For our example the simpler statement is sufficient.

id	*spelling*
23452	"color"
.	.
.	.
.	.

id→spelling

position	*id*
124	23452
.	.
13	23452
.	.
.	.
212	23452
.	.
167	23452

position→id

By splitting the data into two tables, we have separated the spelling of the word from its position in the text. The first table defines the spellings; the second defines the positions of words in the text. The original schema contained a redundancy (the spelling of the word was repeated many times) that is removed in the normalized schema.

Normalizing involves decomposing a schema in such a way that the result expresses the same set of facts, but exhibits more desirable update properties. Under normalization, the explicit dependencies do not change; what changes are the *update constraints*. The original schema for the editor, consisting of id-spelling-position, involves the update constraint that three values must be entered for each new data item. Normalization results in a decomposition that provides two new update constraints—id-spelling and position-id. The new update constraints result in simpler update, while maintaining the same information structure.

Normalizing the schema removes the update redundancy. However, very few editors use normalization; most address the problem of redundancy not by removing the redundancy, but by providing special commands that transfer the redundant work from the user to the editor. Accomplished users of text editors avoid redundant effort by employing global search-and-replace operations to change spelling with a single command. It is interesting to consider some of the problems engendered by this solution. Search-and-replace allows us to easily change all instances of ‘‘color’’ to ‘‘colour’’, but it can overshoot its mark: it may also incorrectly changes instances of trademarks such as ‘‘Technicolor’’ to ‘‘Technicolour’’. It may also incorrectly change embedded references, which should maintain the spelling of the source document, rather than the current one. Search-and-replace commands can also undershoot the mark by missing textual variants, such as capitalized forms of the word, words that are hyphenated or split across lines, or words that include punctuation or typesetting codes (and hence do not match the search pattern). Consequently, authors must use search-and-replace carefully[1].

Boyce-Codd normal form avoids the problems of search-and-replace by eliminating the redundancy, and hence the need for a redundant operation. Why are editors not designed in this fashion, then? In order to use Boyce-Codd normal form directly, users would have to enter text by specifying the *id* of words, rather than their spellings; this is clearly unacceptable. Secondly, users tend to tolerate a certain amount of anomaly in their documents. They do so partly because inconsistent or incorrect spelling is only one of several types of grammatical mistake or inconsistency, and partly because the meaning of the text is usually clear from the context (it is interesting that one redundancy in the text offsets errors in another). Thirdly, for most users the concept of search-and-replace is much easier to understand than is that of Boyce-Codd (or any other) normal form. Finally, most designers of editors see themselves as building tools to work with a specific representation—namely, a text file that exhibits redundancy—rather than building an update tool for a database. The editor `sam` is based on a transaction model, and so takes a first step in the latter direction[24].

Text editing illustrates a general problem: when confronted with data redundancy, one can choose to remove it, or to deal with it through automatic redundant processing. Editor designers have voted solidly for redundant processing, but in other contexts, this choice is made differently. One of the major empirical lessons of database practice is that redundant processing is a short-term solution that has disastrous long-term effects. The reason is simply that over the long term, we want to change the software and hardware environment, but keep the dependencies and the data constant. This is very difficult to do if part of the dependency structure is captured in the applications software. Early database systems did not clearly separate dependencies from the software, and so programmers wrote applications that relied on implementation details and semantic information. The result was a database that worked, but that was also extremely difficult to move to a new operating system or machine. In many cases, it was simply impossible to predict the implications of any systems change, because some of the applications programs even made assumptions about the number of bits allocated to data fields[12].

Similarly, search-and-replace in a text editor takes advantage of some implementation details to achieve a certain effect. This is a thoroughly acceptable technique for small text files that are mostly the province of one user. When text files are large or shared, however, normalization may be a better design choice. Searching a text database for string matches becomes problematic when the database is large collection of text created by numerous individuals over a long period of time[5]. Here, variants in term usage, spelling, and abbreviation make string-based searching unreliable. Text databases employ several means to address this problem, such as controlled vocabularies, centralized abstracting or assignment of keywords, and thesaurus-based searching. Each of these tools in effect applies a normalization to the text, either by restricting word variants, or by organizing a table of variants so that queries can be converted into ‘‘ids’’. Small text processing problems can be handled by redundant processing, but larger ones are more likely to benefit from normalization.

4. Spreadsheets.

One of the most common tools for managing and updating information is the spreadsheet. Existing work on spreadsheets has focused on such issues as the types of errors committed by users, the distribution of command usage, and the extension of the spreadsheet technique to logic and object-oriented data[7, 22, 23, 26, 28]. Spreadsheets are a particularly interesting case for data dependency theory, since much of the spreadsheet user's effort is spent on creating and manipulating dependencies, and much of the spreadsheet's functionality is intended to simplify dependency management.

Spreadsheets are superficially analogous to relational databases, since they both deal with (mostly) numeric data in tables. As a result, we might incautiously assume that spreadsheets could be decomposed in a fashion similar to relational databases. But spreadsheets and relational databases are very different kind of tables. One difference is that each row of a relation must be unique, whereas spreadsheets can have duplicate rows. More generally, the rows and columns of a spreadsheet have very little formal meaning; they serve primarily as a convenient set of placeholders for organizing data, and for use as identifiers in formulas. Users generally organize their sheets in some meaningful way, but the sheet itself does not rely on the semantic aspects of the layout (cell proximity) to compute any of its values. What the sheet does enforce is the set of formulas defined by the user.

We will use the following spreadsheet as an example in our discussion of dependencies:

	A	B	C	D
1			A1+B1	
2			A2+B2	
3			A3+B3	
4			A4+B4	
5		C2-C4		
6	A1*A2			

This sheet contains no values, only formulas. The top four formulas in the sheet define the values of the cell in column C to be the sum of the cells in columns A and B of the corresponding row. The fifth formula in the table derives a result from two of the existing formulas. The sixth formula re-uses values that were used in the first two formulas of the spreadsheet.

One simple functional dependency enforced by all spreadsheets is *cell uniqueness*: every cell in the sheet is the target of at most one formula:

$$col, row \rightarrow formula$$

If a spreadsheet did not enforce cell uniqueness, a single cell could be the target of two or more formulas, leading to an ambiguous result. Spreadsheets enforce this dependency by storing the formula within the cell that it targets. As only one formula can be kept per cell, formula updates will automatically obey this dependency. If the spreadsheet stored formulas outside the table, it would need an additional mechanism to ensure cell uniqueness.

The most important spreadsheet dependencies are *formula dependencies*. These describe the underlying formulas, ensuring that each formula has one set of arguments and operators that defines the result:

$$formula, arg_{1,}\ op_{1,}\ arg_{2,}\ op_{2,} \ldots arg_n \rightarrow result$$

Formula definitions are functional dependencies, because a particular set of arguments identifies a unique result for the formula. Moreover, they are a restricted case of functional dependency, one in which the result can be computed from the arguments. Formula dependencies can be transitive; that is, the result of one function can be an argument for another.

The formula and cell dependencies of our example spreadsheet can be captured by a schema such as the following:

col_1	row_1	op	col_2	row_2	col_r	row_r
A	1	+	B	1	C	1
A	2	+	B	2	C	2
A	3	+	B	3	C	3
A	4	+	B	4	C	4
C	2	-	C	4	B	5
A	1	*	A	2	A	6

Each row in this table corresponds to one cell dependency and one formula dependency. The cell dependency is expressed by making the result cell a key for the table; hence, there is a unique pair (col_r, row_r) for each row. The formula dependency is expressed by using the other parts of the row to compute a value for the (col_r, row_r) cell.

Formula dependencies are involved in several types of update. The first is simple data update, in which the values of some of the arguments are changed and the result of the formulas is recomputed. Spreadsheets are very good at managing this kind of update, partly because no change is made to the schema of formula and cell dependencies; the sheet merely recomputes the result values from the cells. The sheet ensures that transitive dependencies are computed; in our example, if the value of A2 were changed, the sheet would recompute the second formula (A2 + B2), the sixth (A1*A2), and the fifth (C2-C4), since the fifth formula is transitively dependent on the value A2. Data update is exploited by users who investigate ‘‘what-if’’ scenarios, where a hypothetical update to some base value (e.g., sales) can be used to compute the expected effect on some derived value (e.g., profits).

A second type of update is rearranging the layout of the sheet. This usually involves changing the dependency schema, because the arguments are expressed in terms of layout coordinates. It is somewhat paradoxical that data update causes no schema changes, though it does result in new data values, while layout update causes schema changes, even though the data values remain the same.

There are several ways to change the layout of a sheet. One can move or copy data cells, move or copy formulas, move or copy blocks of cells, or insert or delete rows or columns. Most spreadsheets attach different semantics to these operations, attempting to anticipate the user's intention. In Quattro® Pro, for instance, moving a data cell results in updates to the formulas that reference that cell; copying the cell does not result in such an update[2]. Moving data *into* a cell referenced by a formula, however, causes the formula to display an error message, rather than to update the formula (because the sheet assumes that the user is unaware of the cell dependency). Moving a formula results in no change to its cell references, while copying a formula along with its referenced cells will update relative cell references (but not absolute ones). Block moves and copies have additional semantics, depending on whether coordinate cells are changed (in which case the assumption is that the block is being redefined, either correctly or incorrectly).

This tangle of semantics expresses itself as a complicated set of updates to the schema described above. Moving a formula is fairly straightforward; it constitutes updating the (col_r, row_r) values of the corresponding row in our formulas schema. Copying a formula with absolute references is almost the same, except that a new row is created with the new col_r and row_r values. Copying a formula with relative references involves creating a new row with new (col_r, row_r) and values for (col_1, row_1), (col_2, row_2), that are derived from the displacement of the old (col_1, row_1), (col_2, row_2), from the old (col_r, row_r). Copying thus attempts to maintain a dependency between the positions of the arguments and the position of the result. These operations are applied multiple times when moving or copying blocks of cells that contain more than one formula. It is interesting to note that copying a formula and deleting the original may not have the same effect as moving it.

Moving and copying can be confusing operations, so some spreadsheets permit a level of indirection between formulas and their arguments. Rather than referencing cell positions, formulas can reference abstract quantities which are mapped to cell names. For example, instead of defining a formula in terms of cell references:

C2 - C4

it can be defined as

sales - expenses

The mapping between conceptual variables and cells is maintained by an indirection table:

col	*row*	*concept*
C	2	sales
C	4	expenses

If all formulas are expressed in terms of conceptual variables, then the layout of the sheet can be changed without affecting the formulas; the only table that needs to be updated is the indirection table.

The indirection table is a normalization of the formula table (although not one that corresponds to any known normal form). It captures in one place the mappings between the logical relationships of the cells and their physical positions, and so updates to the indirection table can be made without disturbing the formula schema. Storing row and column information directly in the formula table, on the other hand, is not normalized, since moving a single column or row may affect several cells other than the ones in the given row or column.

The indirection table also better expresses a third type of spreadsheet dependency, namely *argument equivalence*. This dependency says that formulas that refer to the same argument should always use the same value. Spreadsheets manage this type of dependency by encouraging users to store a data value in a single place; in our example, the first and sixth formulas share the data cell A1. This would be sufficient for maintaining this dependency, if the spreadsheet's layout were never updated. Updates to the layout can be complex, however, and so it is useful to normalize the formula table and thus make layout updates more robust.

A fourth type of spreadsheet dependency is *operation equivalence*. Operation equivalence is a dependency between two or more formulas which should have the same operations. Formulas that compute an average, for example, may be required to compute a single type of average (e.g., a mean). If the type of average changes during the life of the spreadsheet (e.g., to a mode or median), then all averaging formulas must be located and updated. Spreadsheets not only do not manage this dependency, but actually exacerbate the problem, since they encourage users to make copies of formulas in order to generate equivalent computations. In our example spreadsheet, the sum formula for column C is repeated four times. If these copies are similar not merely by accident, but because it is intended that the same operation be performed for each row, then a redundant update will be required if that operation is changed. In general, any time an interface provides a ‘‘copy’’ operation, we should be on guard for the possibility of redundancy and update anomaly.

Another type of dependency not managed by spreadsheets is *non-formulaic dependencies*. As an example, a personnel spreadsheet may store employees and managers, and it may be the case that every person has only one manager. Since the manager cannot be computed from the value of the person, however, this dependency cannot be expressed in most spreadsheets. Non-formulaic dependencies are common in relational databases, where the user is expected to provide the values for both sides of the dependency.

Complex spreadsheets with many interrelated formulas can be difficult to trace and debug when the cells for a given formula may be scattered about the sheet. It is interesting to note that this problem is a direct result of the choice to normalize data to avoid update anomalies. Consider an alternative spreadsheet design, in which the arguments for each formula are clustered together in a single location and data values are entered redundantly. In our previous example, then, the first and sixth formulas would not both refer to the location A1, but would each have a private copy of the value stored at that location. In order to ensure that the redundant values are consistent, we need a transaction operator that updates them simultaneously. Curiously enough, this alternative design is more or less equivalent to the simple text editor, whose ‘‘values’’ are words, whose ‘‘formulas’’ are sentences, and whose ‘‘transaction operator’’ is search-and-replace. In this alternative, the data is unnormalized, and so users can more quickly find the arguments of the formulas. It is not a coincidence that improving the user’s ability to understand and trace through the sheet involves some type of un-normalizing; users themselves often add redundancy to a spreadsheet, by arranging for certain redundant ‘‘check’’ values to be computed[20]. The tug-of-war between structuring for ease of use and structuring for update control is one reason that database systems support two or more levels of structure. An internal level maintains schemas that are normalized and thus more robust to update, while the external level supports views on these schemas which may be redundant for ease of use.

We have seen that spreadsheets provide a clutch of techniques for managing dependencies and updates. In some cases normalization is used; for example, only one copy of data values is stored. In other cases, redundancy is tolerated because it can be managed by some underlying computation that ensures consistency; for example, a redundant cell reference is automatically managed when the cell is moved to a new point in the sheet. Some uncontrolled redundancy exists; for example, dependencies between formula operations are not captured. In other cases, the user has a choice of normalization or relying on underlying computations; for example, formula arguments may be entered either as row-column values (which are managed to some extent by the layout operations) or they may be abstract labels (managed by an indirection table). It would be interesting to know whether a more effective and comprehensible design could be achieved with fewer methods for managing dependency.

Spreadsheets illustrate three important issues in applying dependency theory to software design. First, it is important to correctly identify the abstract structure of the system before applying the theory. In the case of spreadsheets, we sidestepped the cursory similarity between spreadsheets and relations, focusing instead on the the formulas. Second, it is important to look at all update problems. Spreadsheets optimize some types of data update, but pay less attention to formula update, and no attention to non-formulaic dependencies. Third, normalization has advantages and disadvantages. Update anomalies can be eliminated by normalization, but normalization can also increase the complexity of navigating a structure.

5. Updating software.

Applications often manage data that is updated, and applications are themselves updateable data. It is a folk theorem that the one constant in software design is change—in other words, update. It would not be surprising, then, if some aspects of dependency theory were relevant to the study of software update. The need for normalization, for example, is implicit in the proposals for user interface management systems[25]. UIMS's were intended to improve user interface software by taking the burden of interface interaction away from the individual applications and centralizing it within the interface management system. In effect, this was a recognition of redundancy (each system managing its own interface), and the centralization of this activity (all interfaces handled by one system), with the consequent advantages of consistency and ease of update (all interfaces could be changed at one time). More recently, object-oriented approaches have been touted as providing similar advantages[3].

Our description of software update requires the database notion of *multivalued dependencies*. A multivalued dependency between X and Y, written $X \rightarrow\rightarrow Y$, says that each unique value of X is associated with a set of values of Y. As an example, a team is associated both with a set of coaches and a set of players; hence, $team \rightarrow\rightarrow coach$ and $team \rightarrow\rightarrow player$. The important idea behind this type of dependency is that it captures the cross product of two sets of information: every coach is a coach of every player. Schemas that are robust to updates of data constrained by multivalued dependencies are said to be in *fourth normal form*. Multivalued dependencies are a generalization of functional dependencies, and fourth normal form is a generalization of Boyce-Codd normal form.

Let us return to the text editor, considering it from the viewpoint of a programmer responsible for developing the editor's interface. We assume that the editor operates in a window environment, and hence can be controlled with function keys, pull-down menus, or command lines. Given this variety of input devices, the novice designer constructs the editor in a fashion that is captured by the following schema:

device_instance	*operation_instance*
menu option "delete line"	delete_line
menu option "insert word"	insert_word
fn key F1	append_file
fn key F2	clear_buffer
fn key F3	insert_word
command line "iw"	insert_word

$$device\_instance \rightarrow operation\_instance$$

The novice programmer writes a program in which each input device is directly and independently mapped to an action, as if it were constructed with point-to-point wiring. The program that implements the schema usually resembles the following:

```
switch (event)
{
  menu:    option = get_option();
           if (option == delete_word)
                delete_word();
           if (option == insert_word)
                insert_word();
           break;

  fn_key1: append_file();
           break;

  fn_key2: clear_buffer();
           break;

  fn_key3: insert_word();
           break;

  cmd_ln:  cmd = get_cmd();
           if (cmd == "iw")
                insert_word();
           break;
}
```

This type of design is relatively straightforward, but as can be seen by the sole functional dependency, it does not insist on very much consistency. Modifications to the program (that is, updates to the design) can result in errors or inconsistency. For example, the schema permits operations to be mapped to multiple instances of a device; consequently, the programmer might inadvertently assign more than one function key to buffer clearing. This redundancy makes it harder to consistently change the mapping of function keys to operations. Also, the schema does not require all operations to be accessible from a device. The programmer might easily build a system that allowed the use of menus to delete words, but not to insert them. These kinds of inconsistency are bound to influence both the robustness of the interface and its comprehensibility to the user.

The schema has update anomalies, so we may be able to decompose it to avoid the anomalies. We cannot, however, decompose a schema with only two columns. First, we must decompose the columns themselves. The columns are compositions of atomic values; if we were to place each of these in its own column, then we would be able to further decompose the schema. In database parlance, the schema needs to be converted into *first normal form*, in which each attribute contains a single atomic value. First normal form is usually a prerequisite for achieving further normalization. The first normal form of the editor schema is:

device	*op_label*	*u_label*	*op*	*unit*
menu	"delete"	"line"	delete	line
menu	"insert"	"word"	insert	word
fn_key	F1	-	append	file
fn_key	F2	-	clear	buffer
fn_key	F3	-	insert	word
cmd_ln	i	w	insert	word

Having produced first normal form, we now try to identify the dependencies that exist between the atomic data values. An obvious dependency is that each device and label should perform a single operation. This is merely a restatement of the original dependency:

$$device, op\_label, u\_label \rightarrow op, unit$$

The relationship between units and operations is a complex one. There seem to be groups of units and associated operations that we might call ‘‘blocks’’. One block consists of things like lines and words, which have the associated functions ‘‘delete’’ and ‘‘append’’; another block is files and buffers, which have the associated actions ‘‘clear’’ and ‘‘append’’. There is not enough data in the instance to verify this model, but it seems reasonable to assume that these classes could be extended to include both more operations and more units. In general, then we expect the following multivalued dependencies to hold:

$$block \rightarrow\rightarrow op$$
$$block \rightarrow\rightarrow unit$$

Similarly, we observe that there are several input devices. It seems reasonable that users should have consistent access to the operations from each of the devices. Users who prefer menus, for example, would not like a system in which they could use the menu to delete words, but had to use a function key to delete characters. This suggests the following dependency:

$$block \rightarrow\rightarrow device$$

Thus ‘‘block’’ is an entity that emerges when we consider the implications of extending our schema to cover data values that are not present in the given instance. Update often adds new data values, so it is important to consider the range of potential updates when normalizing a design, and not just the values that are present at design time.

Given the above dependencies, we can decompose the relation into fourth normal form. The first schema defines the set of devices that can access each block:

device	*block*
menu	1
fn_key	1
menu	2
fn_key	2
command line	1

The second schema defines the set of operations that belong to each block:

block	*op*
1	delete
1	insert
2	clear
2	append

The combination of the first two tables ensures that each device that can access a given block has access to all its operations. Hence, we can add or remove operations consistently across devices, and add or remove devices across operations. The third schema defines the set of units that can be modified by operations in a block:

block	*unit*
1	character
1	word
1	line
1	paragraph
2	file
2	buffer

The three schemas ensure that operations of a block are applicable to each of the units of that block. Thus we can add or remove units consistently across operations, and similarly add or remove operations consistently across units.

The fourth and final schema defines the set of labels that are used for each operation and unit:

device	*o_label*	*u_label*	*op*	*unit*
menu	"delete"	"word"	delete	word
menu	"insert"	"word"	insert	word
fn_key	F1	-	append	file

Experienced programmers will tend to develop software that embeds the second schema above. The resulting program might look like the following:

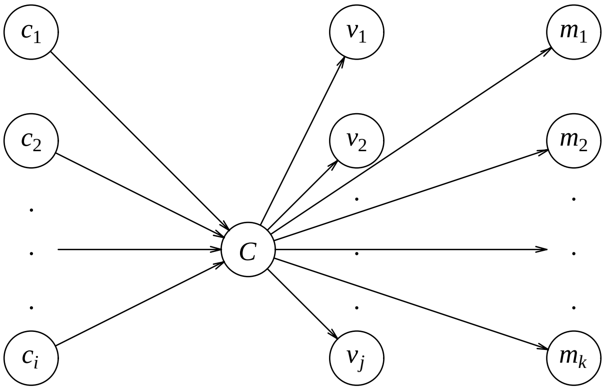
```
switch(block)
{
     case text:
       if (op == insert)
          insert(text_unit);
       if (op == delete)
          delete(text_unit);

     case buffer:
       if (op == clear)
          clear(buffer_unit);
       if (op == append)
          append(buffer_unit);
}
```

The entity ‘‘block’’ has simplified the management of a large group of operations, units, and devices. Whereas the novice program used a top-level decomposition based on devices, the new program is oriented towards the type of object being edited. As a result, the new program clusters the operations and units in such a way that they can be automatically checked (e.g., that no buffer operations are applied to text units). Furthermore, the second program supports a wider range of commands with less code. In the naive interface, there are k operations, m operands, and p devices, hence kmp individual statements may be necessary to make the connections. If the multivalued dependencies implicit in the system are exploited, however, then the number of connections is only $k+m+p$, since their cross product is given by the dependency.

Readers familiar with object-oriented systems will have recognized that ‘‘block’’ is a class with an associated set of methods (*operation*) and a defined set of variables (*unit*). A large part of the structure of an object-oriented system is its representation of multivalued dependencies (although few object-oriented systems designers think about their designs in this way).

Formally speaking, an object type is a mapping between a set of instances, a set of methods, and a set of variables. This mapping is known as a class definition. All instances c_i of class C exhibit methods m_i and variables v_i:



The class relationship expresses the following multivalued dependencies:

$$C \rightarrow\rightarrow m$$
$$C \rightarrow\rightarrow v$$
$$C \rightarrow\rightarrow c$$

If a new instance is created, it must possess each of the methods and variables; if a new method is added, it must be applicable to each of the instances and variables; if a new variable is added, it belongs to each of the instances and allows access by any method.

Object-oriented systems use their (implicit) multivalued dependencies to validate and automate the programming process. Validation occurs because references to variables or methods that are not part of the class's definition (or of its inheritance structure) can be rejected. Automation occurs because the addition of new classes and methods permeates the dependency structure to affect all relevant instances, without the need for the programmer to explicitly manage this operation.

In most object-oriented systems, the multivalued dependency that defines a class is part of a larger set of dependencies known as the inheritance structure. An individual class inherits methods and class variable from its parent objects, and defines methods and classes that are inherited by its subclasses. Updates to inherited methods and variables thus affect not only instances of a class, but (potentially) its subclasses as well.

The automatic support of multivalued dependencies provided by object-oriented systems simplifies some design problems, since it moves the dependencies from control flow into class definition statements. With the proper class definitions, the control flow for the editor interface might look like the following:

```
unit->op(arg);
```

Here the unit (e.g., *paragraph*) is sent a message to perform some operation (e.g., *add*) to itself with an argument *arg* (e.g., a text string). Constraints on the set of valid operations are controlled by the class definition. On the other hand, the normalization of information can also make comprehension more difficult, as it does in spreadsheets; in order to understand a given class, the programmer may need to consult widely dispersed information. An object-oriented class structure has the additional complication that a static determination of the unnormalized structure may not be possible; if the method to be executed is dependent on the class of the argument object, then a run-time calculation is necessary.

In the object-oriented approach, entities and their relationships are explicit, while the dependency structure is implicit. This is similar to database's entity-relationship model[8]. In the relational model, on the other hand, entities and relationships are implicit, and the dependency structure is explicit. A relational schema consists of relations and dependencies; tuples are not required to represent entities, and the relations can be restructured as desired, so long as the dependencies are maintained. The lack of entity-relationship structure in the relational model is an advantage in situations where the identification of entities is contentious or impractical[12]. Our description of object-oriented systems as a means for managing multivalued dependencies is a step towards making dependencies explicit. From the point of view of dependency theory, what is important about inheritance is its implicit recognition and normalization of multivalued dependencies. Increased reuse is a side benefit of normalization; the common elements of code are factored out to be reused, instead of being replicated. This view suggests that it may be possible to describe object-oriented systems in an ‘‘object-free’’ fashion. Rather than teaching programmers that the inheritance structure expresses ‘‘is-a’’ relationships (with the attendant ontological conundrums), it may be useful to teach that inheritance is a way of managing large collections of multivalued dependencies.

6. Discussion.

We have presented only the basics of dependency theory, with an eye to persuading designers that there is some value in using database technology in their own work. Dependency theory and normalization can help to root out the update problems implicit in the design of an interface, and the schemas are a starting point for data structure implementations.

The text editor problem showed that, given redundancy, we can choose between normalizing a schema or providing functionality that transfers the redundancy from the user to the application. The choice is based on the need for long-term consistency of the data. We should note, however, that other update anomalies (such as loss of data and the need to enter ‘‘too much data’’) can only be effectively addressed by normalization.

The spreadsheet problem showed that a single application can employ combinations of normalization and redundant processing techniques to maintain dependencies on data and functions. The result can be bewildering to the user, who must deal with a variety of techniques that have both logical and operational effects.

The editor design problem showed that we can extract dependencies embedded in the control flow of an application. Part of the benefit of object-oriented systems is that they tend to make these dependencies explicit in class descriptions. Just as in the case of spreadsheets, however, this simplifies the task of program update while complicating the task of debugging; tracing back through the dependency structure requires good tools for browsing the class library.

It would be ideal if database conceptual modelling techniques were sufficiently simple and generally applicable that they could easily be applied by system designers. That is certainly our long-term hope, but at present, most people find conceptual modelling difficult, and prefer to design intuitively. Intuitive design is often a quick and appropriate route to systems; but if designers are unsatisfied with their systems, they may find it useful to consider them more formally. Judging by the difficulty many people have in developing good object-oriented class libraries, formalization may turn out to be an important aid.

We have not shown that all software systems are reducible to database systems. That view, even if complete, is just as impoverished as the view that all systems are reducible to editors or Turing machines. Each framework emphasizes a particular aspect of the phenomena at hand, and gains both its advantages and disadvantages from this one-sidedness. What we have shown is that the update implications of systems are important, and that they can be profitably studied in a formal setting taken from dependency theory.

Dependency theory as applied to software engineering resembles viscosity, the notion that systems are resistant to change[15]. Both viscosity and dependency theory are concerned with update and obstacles to update. However, they differ in that viscosity attempts to subsume implementation, interface, task, and user, while dependency theory captures only the abstract schema of the system. Also, viscosity is largely a metaphorical concept, where dependency theory is a formal one.

There are several directions for further work on the application of dependency theory to software engineering. The theory itself is lacking in its ability to describe some types of dependencies, such as those exhibited by sequential or transitive structures. Some of the spreadsheet dependencies and normalizations have no analogue in database dependency theory. A more comprehensive theory would benefit database designers as well as system designers. A second area for work is in the development of automatic tools for dependency recognition. Such tools might make the use of theory more palatable, and so might encourage designers to pay attention to the dependencies and update properties of their interfaces. One example is the automatic extraction of dependencies from forms[9, 10]. A third area of work is a more general study of the notions of update, dependency, and normalization as core aspects of any type of computation. Database dependency theory provides a hint of the connection and the generality of these concepts, but we do not yet understand the fundamental relationships between them, or the laws that govern their interaction. A theoretical investigation of this area will lead to new directions for conceptual modelling.

7. Acknowledgments.

Our thanks to Mert Cramer, Bonnie Nardi, and Thomas Green for their thoughtful comments on the many early drafts of this paper. This work was financially supported by an IBM Canada Research Fellowship and by the Natural Sciences and Engineering Research Council of Canada.

8. References

1. *Chicago Guide to Preparing Electronic Manuscripts for Authors and Publishers*, University of Chicago Press, Chicago, Illinois (1987).

2. Borland International, Inc., *Quattro® Pro Version 2.0 User's Guide*. 1990.

3. ''Special Issue: Object-Oriented Design,'' *Communications of the ACM*, **33**(9)(September 1990).

4. Catriel Beeri, Philip A. Bernstein, and Nathan Goodman, ''A Sophisticate's Introduction to Database Normalization Theory,'' *Proceedings of the 4th Conference on Very Large Data Bases*, (1978).

5. David C. Blair and M.E. Maron, ''An Evaluation of Retrieval Effectiveness for a Full-Text Document-Retrieval System,'' *Communications of the ACM*, **28**(3) pp. 289-299 (March 1985).

6. Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, (eds.), *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer-Verlag, New York, N.Y. (1984).

7. Polly S. Brown, John D. Gould, and An Experimental Study of People Creating Spreadsheets, *ACM Transactions on Office Information Systems*, **5**(3) pp. 258-272 (July 1987).

8. Peter Pin-Shan Chen, ''The Entity-Relationship Model — Toward a Unified View of Data,'' *ACM Transactions on Database Systems*, **1**(1) pp. 9-36 (March 1976).

9. Joobin Choobineh, Michael V. Mannino, and Veronica P. Tseng, ''A Form-Based Approach for Database Analysis and Design,'' *Communications of the ACM*, **35**(2) pp. 108-120 (February 1992).

10. Joobin Choobineh and Santosh S. Venkatraman, ''A Methodology and Tool for Derivation of Functional Dependencies From Business Forms,'' *Information Systems*, **17**(3) pp. 269-282 (1992).

11. E.F. Codd, ''A Relational Model of Data for Large Shared Data Banks,'' *Communications of the ACM*, **13**(6) pp. 377-387 (June 1970).

12. E.F. Codd, *The Relational Model for Database Management: Version 2*, Addison-Wesley, Reading, Massachusetts (1990).

13. Alan F. Dutka and Howard H. Hanson, *Fundamentals of Data Normalization*, Addison-Wesley, Reading, Massachusetts (1989).

14. Michael T. Garrett and James D. Foley, ‘‘Graphics Programming Using a Database System With Dependency Declarations,’’ *ACM Transactions on Graphics*, **1**(2) pp. 109-128 (April 1982).

15. T.R.G. Green, ‘‘The Cognitive Dimensions of Viscosity: A Sticky Problem for HCI,’’ *Proceedings of INTERACT ’90 IFIP Conference on Human-Computer Interaction*, pp. 79-86 (September 1990).

16. L. Kerschberg, A. Klug, and D. Tsichritzis, ‘‘A Taxonomy of Data Models,’’ *CSRG-70*, Computer Systems Research Group, University of Toronto, (May 1976).

17. Haim Kilov, ‘‘Information Modeling Concepts and Guidelines,’’ SR-OPT-001826, Bell Communications Research (January 1991).

18. E.W. Mackie, G.M. Pianosi, and G. de V. Smit, ‘‘Rita—An Editor and User Interface for Manipulating Structured Documents,’’ *Electronic Publishing—Origination, Dissemination, and Design*, **4**(3) pp. 125-150 (September 1991).

19. David Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland (1983).

20. Bonnie A. Nardi and James R. Miller, ‘‘Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development,’’ *International Journal of Man-Machine Studies*, **34** pp. 161-184 (1991).

21. David Notkin, ‘‘Interactive Structure-Oriented Computing,’’ CMU-CS-84-103, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania (February 1984).

22. Judith R. Olson and Erik Nilsen, ‘‘Analysis of the Cognition Involved in Spreadsheet Software Interaction,’’ *Human-Computer Interaction*, **3**(4) pp. 309-349 (1987-1988).

23. Kurt W. Piersol, ‘‘Object Oriented Spreadsheets: The Analytic Spreadsheet Package,’’ *Proceedings of OOPSLA ’86*, pp. 385-390 (November 1986).

24. Rob Pike, ‘‘The Text Editor sam,’’ *Software—Practice and Experience*, **17**(11) pp. 813-845 (November 1987).

25. Jarrett K. Rosenberg, Ralph Hill, Jim Miller, David Shewmake, and Andrew Schulert, ‘‘UIMSs: Threat or Menace?,’’ *Proceedings of the CHI ’88 Conference on Human Factors in Computing Systems*, pp. 197-200 (May 15-19, 1988).

26. Jorma Sajaniemi and Jari Pekkanen, ‘‘An Empirical Analysis of Spreadsheet Calculation,’’ *Software—Practice and Experience*, **18**(6) pp. 583-596 (June 1988).

27. Jeffrey Alan Scofield, ‘‘Editing as a Paradigm for User Interaction,’’ Ph.D. Thesis, Department of Computer Science, University of Washington, Seattle, Washington (August 1985).

28. Michael Spenke and Christian Beilken, ‘‘A Spreadsheet Interface for Logic Programming,’’ *Proceedings of the CHI ’89 Conference on Human Factors in Computing Systems*, pp. 75-80 (April 30—May4, 1989).

29. Tim Teitelbaum and Thomas Reps, ‘‘The Cornell Program Synthesizer: A Syntax-Directed Programming Environment,’’ *Communications of the ACM*, **24**(9) pp. 563-573 (September 1981).

30. Jeffrey D. Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Rockville, Maryland (1988).