A COMPARISON OF SELF-CONTAINED AND EMBEDDED DATABASE LANGUAGES.

A. Christensen and T.U. Zahle

Dept. of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 København Ø, Denmark

ABSTRACT.

The purpose of this paper is to compare self-contained and embedded database languages. An overview is presented, summarising the differences between the two types. It is observed that the principle differences stem from the fact that many things are prespecified in the self-contained languages. It is then argued, that these prespecifications can be carried over to the embedded languages, thereby improving compactness. These embedded languages should also contain the possibility of overriding the prespecifications so that the flexibility of them is preserved. Finally, two examples of such improved embedded query languages are given and illustrated with examples.

1. Introduction.

Usually there are two ways of accessing a relational database, either through a selfcontained language in an interactive dialogue, or through commands embedded in a high-level language like COBOL or PL/I.

The self-contained languages are often introduced as a tool for the ordinary user, as a simple and easy-to-learn high-level language. The embedded languages on the other hand are meant for programmers and other computer oriented professionals.

For the user of the self-contained language, the possibilities of manipulating data from the database is restricted to what is offered by this language, while the programmer using the embedded language, as well as having the possibilities of the database commands, is given the often much wider possibilities of the host language.

From a number of practical projects we have experienced that the self-contained languages are rarely able to fulfill the needs of the user - they lack sufficient flexibility, even with the use of common aggregate functions.

On the other hand even a simple query is of a considerable size in an embedded language, and the starting point of this investigation has been an attempt to pin down the reasons for this size.

2. A comparison between self-contained and embedded languages.

Two languages were chosen for the comparison, QUEL/EQUEL from the INGRES database system, and SQL/ESQL which is used in several different database systems. (We introduce here the term ESQL for the embedded version of SQL).

INGRES with QUEL was developed at the University of Berkeley for the UNIX operating system in 1973, and in 1974-75 an embedded version of the query-language was introduced.

The first SQL version was developed for the System R database at the IBM Research Laboratory in San Jose, with the name SEQUEL2. Like QUEL, SEQUEL2 was developed as a self-contained language, but an embedded version soon followed, first in PL/I, and later in a number of other programming languages. SQL is today recommended for standardization in ISO, both as a self-contained language and as an embedded language for a number of standard programming languages [ISO85].

Both SQL/ESQL and QUEL/EQUEL contain data-definition and data-manipulation, offering the possibility of mixing definition and manipulation within the same session.

```
SELECT S#, QTY
    FROM SP
    WHERE P# = 'P4'

result: **************
    *  S# * QTY *
    **************
    *  S1 *  100 *
    *  S3 *  150 *
    *  S5 *  100 *
    **************

    UPDATE SP
        SET QTY = QTY*1.1
        WHERE P# = 'P4'
            AND S# IN ('S1','S3')
```

Fig. 2.1. SQL

```
1  SQLEX:PROC OPTIONS (MAIN);
2      EXEC SQL BEGIN DECLARE SECTION;
3          DCL S#    CHAR(5);
4          DCL ANSWER CHAR(3);
5          DCL QTY   FIXED BINARY(31);
6      EXEC SQL END DECLARE SECTION;
7
8      EXEC SQL INCLUDE SQLCA;
9      EXEC SQL DECLARE Z CURSOR FOR
10         SELECT S#, QTY
11             FROM SP
12             WHERE P# = 'P4'
13         FOR UPDATE OF QTY;
14
15     EXEC SQL OPEN Z;
16     IF SQLCODE NOT = 0 THEN
17         GO TO QUIT;
18     DO WHILE (SQLCODE = 0)
19         EXEC SQL FETCH Z
20             INTO :S#, :QTY;
21         IF SQLCODE = 0 THEN
22         DO;PUT SKIP LIST('S#=',S#,' QTY=',QTY);
23             GET LIST(ANSWER);
24             IF ANSWER = 'YES'    THEN
25             DO;EXEC SQL UPDATE SP
26                 SET QTY = QTY*1.1
27                 WHERE CURRENT OF Z;
28                 IF  SQLCODE NOT = 0 THEN
29                     PUT SKIP LIST ('UPD-ERROR');
30             END;
31         END;
32     END;
33
34     IF SQLCODE NOT = 100 THEN
35         PUTSKIP LIST('SQLCODE=',SQLCODE);
36     ELSE
37     DO; EXEC SQL CLOSE Z;
38         EXEC SQL COMMIT;
39     END;
40 QUIT: RETURN;
41 END;
```

Fig. 2.2. ESQL - embedded in PL/1

In order to make compare the two types of languages, we refer to a number of problems, which represent the most common types of database queries and manipulations. They are shown and discussed in detail in [CHRI86], and a typical example is shown on the following pages.

The example we use here solves the following problem, for the well known suppliers - parts database :

Find S# and QTY for shipments with P# = 'P4', and ask the user whether to increase some QTY with 10%. (Here shipments with S# = 'S1' and 'S3' are increased.)

Comments to the ESQL solution.

Primarily ESQL makes it possible to perform searching and updating in a single pass. The program steps are as follows:

line 8 :	A communication area (SQLCA) is included in the program.
lines 9-13 :	The actual query is stated here in the cursor declaration.
line 15 :	The query is performed here, when the cursor is opened.
lines 16-17:	Error conditions are checked.
lines 18-32:	A loop is performed once for each selected row, unless an error occurs (SQLCODE ≠ 0). The values of the fields in each row are put in the variables S# and QTY by the FETCH operation, the values are shown, and it is decided if the row is to have its quantity increased. If 'YES' the increase is performed at once in the UPDATE operation in line 25. The UPDATE statement does not identify the shipment row by the usual combination of part number and supplier number, but uses the cursor Z, which is already pointing at the row, by specifying 'WHERE CURRENT OF Z'. This special ESQL feature can used when the 'FOR UPDATE OF columnames' is specified. (Note that ESQL makes it possible to perform the searching and the update in a single pass instead of having to solve the problem in two passes as in SQL.)
lines 34-39:	If all the rows have not been processed properly (SQLCODE = 100) then an error message is printed. Otherwise the cursor is closed and the changes committed.

```
RANGE OF SUPPART IS SP
RETRIEVE (SN = SUPPART.SN, QTY = SUPPART.QTY)
    WHERE SUPPART.PN = "P4"

   result:     *****************
               *  SN  *  QTY  *
               *****************
               *  S1  *  100  *
               *  S3  *  150  *
               *  S5  *  100  *
               *****************

REPLACE SUPPART (QTY = SUPPART.QTY*1.1)
   WHERE  SUPPART.PN = "P4"
   AND (SUPPART.SN = "S1" OR
        SUPPART.SN = "S3"   )
```

Fig. 2.3. QUEL

```
1  MAIN()
2  {
3  CHAR ANSWER (3);
4  ##INT INDX, QTY;
5  ##CHAR SNRTAB[25][6], SNR[6], CMPSNR[6];
6  ##INGRES "-210" ANKERSDB
7  INDX = 0;
8  ##RANGE OF SUPPART IS SP
9  ##RETRIEVE (SNR=SUPPART.SN, QTY=SUPPART.#QTY)
10 ## WHERE SUPPART.PN = "P4"
11 ## {
12    PRINTF("SNR=%S,QTY=%D\N",SNR,QTY);
13    SCANF("%S",ANSWER);
14    IF (ANSWER[0] == 'Y')
15        STRCPY(SNRTAB[INDX++],SNR);
16 ## }
17 WHILE (INDX-- > 0)
18 {
19    STRCPY(CMPSNR,SNRTAB[INDX]);
20 ## REPLACE SUPPART (#QTY=SUPPART.#QTY*1.1)
21 ##       WHERE SUPPART.SN = CMPSNR
22 ##          AND SUPPART.PN = "P4"
23 }
24 }
```

Fig. 2.4. EQUEL - embedded in C

Comments to the EQUEL solution.

In EQUEL you have to separate the operation into a query and an update, just as in SQL and QUEL.

The reason why the EQUEL program does not have the same logical and straight-forward structure as the ESQL program is because it is not possible to mix host language and EQUEL operations in the same block. If this was possible, then lines 20 through 22 could replace line 15, performing the update inside the query-loop as in SQL. (This is unfortunately not possible in the used INGRESS version,and it is a serious limitation.)

lines 8-10 :	The query is stated, exactly as in QUEL.
lines 12-15:	This block of C statements is executed for each selected row and the contents of the row is written onto the output file. A dialogue with the user determines whether the user wants to increase the shown shipment row or not. If so, the supplier number is kept in a field in the C variable SNRTAB, by the operation STRCPY.
lines 17-23:	After having shown all the selected rows to the user, the actual update is performed by looping. For each supplier number in SNRTAB (but not more than 25) a REPLACE operation is performed, updating the quantity of the specified row.

Finally it should be noted, that the character '#' appearing immediately before the column name QTY in the lines 9 and 20, is used to indicate that this is an INGRES column name and not the C variable of the same name. (This marking is only necessary when variable names and column names are alike).

Summing up the examples.

The SQL/ESQL examples show that it is possible to create an embedded language in such a way that the host language expands the possibilities of acting on the results, thus showing the power of the embedded language.

These examples, along with a number of others discussed in [CHRI86], make it apparent that the number of statements needed to write programs in embedded languages is much more than in the similar self-contained languages. The self-contained langauges are therefore often much more compact.

On the other hand, this very example shows a problem that is hard to solve in self-contained languages, since it is both a query to gain knowledge about the database, and an update of certain tuples from the answer.

It is thus apparent that the advantage of self-contained languages is their compactness, while the advantage of embedded languages is their power of expression. Would it be possible to combine the advantages of both?

3. A schematic overview of the typical tasks performed in a database program.

To analyze these examples, we have divided the programs into their basic tasks. The basic tasks are all actions to be performed, directly or indirectly, when accessing the database.

It has been our intention to make a list of tasks that cover all the actions to be performed by the user when creating a program, whether they are explicitly or implicitly specified. This includes tasks prior to writing the actual program, and the tasks necessary to execute the program.

This list of tasks makes it possible to describe the differences between self-contained and embedded languages . The list has been ordered in the sequence in which a session on the database usually takes place. This means that "logon" and database identification are performed first, and saving and "logout" are last.

Along with the list of tasks, follows a classification which shows how the task is triggered in both self-contained and embedded languages. The task can be either explicitly specified in the program, or can be implicitly prespecified.

	emb. lang.	imp. emb. lang.	lang.
login (user-id, password)	s	s	s
monitor-start (monitor-name)	s	s	s
DB-open (DB-name)	s	s	s
editor-start	p	p	s
program-start	-	p	s
cursor declaration / open / close	-	s	s
input-parameter decl/assignment	-	s	s
intermediate-variable decl/assign.	-	p	s
transaction-start	p	p	s
table creation / deletion	s	s	s
DB-oper: selection	s	s	s
function	s	s	s
printing	p	p	s
ok-reaction: handling	p	p	s
message	p	p	s
error-reaction: handling	p	p	s
message	p	p	s
transaction-end	p	p	s
program-end	-	p	s
saving program	s	s	s
editor-end	p	p	s
precompile	p	p	s
compile	p	p	s
link	p	p	s
load	p	p	s
execute	p	s	s
DB-close	p	p	p
monitor-logout	s	s	s
logout	s	s	s

specified='s', prespecified='p' and unused='-'.

Fig. 3.1 Typical tasks for database programs.

Comments to figure 3.1:

First let us explain what is meant by the tasks listed in the figure.

Login and monitor: The first two and the last two are obvious, since login, monitor-start, monitor-end and logout are almost always explicitly defined in a session with a computer.

Datebase: The DB-open is a statement that identifies which database is referred to in the statements to come. In some systems it is possible to disconnect one database and connect another without leaving the monitor. Thus we have placed the DB-close before the monitor-end.

Editor: Usually a standard editor is used when creating a program with embedded statements, without any connection to the database. Conversely the self-contained statements are entered after having entered a monitor. These statements are kept in a buffer, until they are finished and executed.

Program: After having entered the editor, (specified explicitly or implicitly), the program is typed in. Host language programs with embedded statements are in our examples always started with one or two standard statements, which we will call program-start. In self-contained languages there is usually no required program start. The program-end must be stated in both EQUEL and ESQL.

Cursor: Another characteristic of embedded languages is the cursor, which is explicitly declared, opened and closed in embedded SQL, while in EQUEL it is declared, opened and finally closed through a single statement. The cursor is not used in the chosen self-contained languages.

Parameters and Variables: Variables as input-parameters (ANSWER in the examples) or as intermediate variables (QTY, SN and S# in the examples) is another feature of embedded languages. In both of the embedded languages these variables must be declared in specially marked statements, and also marked in the embedded database statements. Variables such as these are not available in usual self-contained languages.

Transactions: The next tasks are expressing action towards the database, but operations that change the database must be put into transactions if using ESQL. Here the COMMIT is the end of one transaction and the beginning of the next. The first transaction is started by the first updating statement. This is not necessary in a self-contained language, since each single statement can be regarded as a transaction,(unless it is overruled by an explicit transaction- start and transaction-end pair enclosing a number of statements). Consequently transaction-start and transaction-end can be classified as prespecified in selfcontained languages, while only transaction-end is specified (explicitly) in embedded languages.

Selection: The next step is to access the data of the database. These are the actual database operations, and they have been subdivided into a number of tasks. First is the selection of the data to be operated upon.

Function: Next is the function, which is the action performed on the data selected. In the example in section 1, the function is the update of the selected rows in the SP-table.

Printing: Next is the intermediate storing or printing of the selected data. In self-contained languages the most common is the printing of data in a prespecified format, while in embedded languages the selected data is usually stored in host language variables.

Error-reaction: Another important part of the selection is the specification of reactions on successful and erroneous operations. Both the ok- reaction and the error-reaction can be subdivided into the action taken and the message given. In the self-contained languages the error-message is usually an error-code, perhaps followed by an explanatory text, and the ok-message is a count of the rows updated, inserted, deleted or selected. In embedded langages as ESQL and EQUEL it is necessary to check status-codes in order to discover errors. Furthermore the errors must be reacted upon, either by returning messages to the user, or by programming other database manipulations.

Saving: After having entered the program with the editor, you might wish to save it for later use, especially if it is a program for general use. This mainly concerns programs with embedded statements, but some programs in self-contained language might also be useful to keep. The way SQL and EQUEL work, it is expedient to keep the program at least temporarily, because of the separation between editing, compiling and running, and both of the self-contained versions offer a possibility of keeping programs in a library.

Execution: After having saved the program, it is time for executing it. This includes compiling (perhaps including a precompiling), linking/loading and finally execution. These tasks are usually performed transparent to the user when executing self-contained programs, but when using embedded languages the user must specify each of the steps.

4. Conclusions about the overview.

A look at the schema in fig. 3.1 shows many differences between self-contained and embedded languages, although the basic constructs of the databaselanguages are the same. The example in the first section showed similarity in the way the selection and update constructs were specified, but never-the-less the number of lines necessary to solve the problem in the embedded languages far exceeded the number of lines used in the selfcontained languages.

In self-contained languages there are no extra surroundings or declarations, and there are no statements for printing or reading, so what is left is the basic operations on the database. In embedded languages a lot of work is done starting, ending and controlling the flow of the program, declaring and initializing variables and printing and reading the dialogue with the user.

The reason for this is that a number of things that are explicitly specified in embedded languages are prespecified in self-contained languages. The concept of variables is unknown in self-contained languages, the only media for temporary storing of data is the user's head or paper and pencil. The signalling of errors or of ok-messages is standardized in a way that makes it unnecessary to express anything about these reactions in self-contained languages. Embedded languages offer only a status-code for internal use in the host program, which makes it necessary to specify in detail any reaction that has to do with the status code, even when it is only to show it to the user of the program.

Another big difference is how the actions are performed in order to execute the typed statements. In self-contained languages you can execute the statements directly after entering them, without really knowing what goes on behind the screen, such as compiling or interpreting. In embedded languages you have to perform a number of actions in order to execute the program. (Of course it is possible to make a standard macro to perform precompiling, compiling, linking, loading and execution of the program, but in order to write this macro, you have to know the basic process.)

The conclusion that can be drawn from this example (and others), must be that the basic difference between self-contained and embedded languages is the number of actions that are either prespecified or unnecessary in self-contained languages. In embedded languages the rule is, that there is no standard solution to anything, so you have to specify everything yourself.

The reason for this difference is, that the self-contained languages are meant for the user who is not a computer professional, while the embedded languages are for the programmer, who wants to control everything himself. Consequently the self-contained languages are kept as simple as possible, by a great number of prespecifications, while the main goal in embedded languages has been to offer as many possibilities as possible, with much less thought for simplicity.

Is it possible to combine the two philosophies into a single language, offering at the same time the simplicity of the self-contained languages and the wide variety of possibilities of the embedded languages? Why not let the prespecifications and standardizations of the self-contained language survive in the embedded languages, *leaving it open to the programmer to overrule the prespecifications?!*

In order to explain how this combination could change the embedded language, the overview from fig. 3.1 is used. It has now been given a column showing what an improved embedded language could look like.

This figure shows the kind of changes we feel are necessary to improve embedded languages. It must be noted, that a "p" in this figure in general means that there is a prespecified handling of the task, but the prespecified handling can be overruled by the programmer.

5. Improving an embedded language.

The next question is how to introduce a higher level of prespecification in an embedded language. We have seven targets for simplification:

A: Program surroundings.
B: Variables.
C: Cursors.
D: OK-reactions
E: Error-reactions.
F: Transaction handling.
G: Program execution.

A. Program surroundings.

Target A is perhaps a minor problem, but if an embedded language is to be simple and short, there is no reason to have to start and end every program with two or three extra lines, when the precompiler might as well put them there. In case a name is wanted for the program, it must be possible to specify a name in the beginning of the program. For instance:

```
PROGNAME (Name).
```

B. Variables.

Target B is somewhat more tricky. It should be possible to refer directly to the fields in the current row, when using a tuple-at-a-time facility such as the cursor-loop. This is not possible in ESQL or EQUEL, but the ability to refer directly to the fields would reduce the number of variable-declarations and assignments considerably, since many variables are only used for printing, or for storing in other tables. In [ZAHL78] the language SCAN was introduced with this facility, and the implementation of the language shows the viability of this facility.

C. Cursors.

Target C is connected to target B, since the cursor is the tuple-at-a-time construct which passes through all the selected rows, returning field-values to the program.

Our wish for improvement is based on the RETRIEVE-statement in EQUEL and on the SCAN-loop in [ZAHL78]. It is desirable that the expression of the selection criteria should be located as close as possible to where the selected rows are used. A way to do this is to let the cursor-specification be the beginning of the loop, and to introduce a statement, END-CURSOR, to end the loop. The statements performed for each of the selected rows will be the statements between the cursor-specification and the END-CURSOR. For example:

```
CURSOR C SELECT S#, QTY
        FROM SP
        WHERE P# = 'P4';
...
...
END-CURSOR;
```

In cases where the cursor-loop is not needed, such as in a simple retrieval of data to be presented in a standard format, it should be possible to specify a selection without a cursor. In this case the rows selected should be written to the standard-output in the format known from SQL and QUEL. For example:

```
SELECT S#, QTY
FROM SP
WHERE P# = 'P4';
```

D. OK-reactions

Concerning ok-reactions there are no reactions at all in ESQL and EQUEL, leaving it to the surrounding program to inform the user that the operation was successful. In a simple selection, presentation of the data is sufficient, but operations such as delete and update should finish with a standard message showing how many rows were changed or deleted, just like the messages given in SQL and QUEL today. In some cases such messages are not wanted, and in those cases they can be excluded by specifying

```
EXCLUDE OK-MESSAGES
```

in the beginning of the program. Then all ok-responses must be specified in detail in the program.

E. Error-reactions.

Concerning error-reactions it is always necessary to return serious error-messages to the user, but usually the error-code will be sufficient to enable the user to correct the error. If the standard error reactions are not sufficient, a way of disabling the standard reactions could be the inclusion of the status-communication-area, in SQL called SQLCA. When this area is included by specifying

```
INCLUDE SQLCA
```

in the beginning of the program, the error-reactions must be explicitly expressed in the program. They can be programmed as a test of the status-parameter SQLCODE, followed by an action depending on the status. The action could be informative to the user (to help him correct the error).

First of all the inclusion of the communication area SQLCA will disable all standard error-reactions, and leave everything to be specified explicitly. Then it is possible to use the handling from the ISO-standard, giving general error procedures for the entire program by specifying:

```
WHENEVER SQLERROR... and
WHENEVER NOT FOUND...
```

or testing directly on the variable SQLCODE after every database operation, giving a special exit-handling for each operation.

F. Transaction handling.

Target F has only little impact on the size of the programs, but it will ease the writing of small uncomplicated programs, since these programs often need no transaction handling. The standard transaction handling should be that the running of a program is treated as one transaction, no matter how many updating state- ments there are in the program. This would mean that if an error occurs in one of the updating operations during the execution of a program, none of the operations are performed. It also means that if the operations of a program are to be separated into several transactions, it should be possible to disable the standard transaction handling, simply by explicitly issuing the statement

TRANSACTION COMMIT

after the last operation in each transaction. The TRANSACTION COMMIT is also the beginning of a new transaction, and if this transaction is not finished explicitly by a TRANSACTION COMMIT, the standard transaction handling will ensure that the last operations are committed when finishing the program.

G. Program execution.

The previous points have all been focusing on the contents of the program itself, but the facilities concerning entering, compiling and executing the programs are also issues where the techniques of the self-contained languages could be introduced for the embedded languages. It should only be necessary to issue a single execute-statement, without having to know about precompilation etc. This could easily be handled by the supervising monitor.

Altogether these changes would give rise to a language combining the compactness of the self-contained languages with the flexibility of the the embedded languages.

6. An improved Embedded Query Language, EQL.

Let us first demonstrate how the improvements described will change the embedded solution to the problem in section 2. First we show an example of how to solve the problem in a language that is different from ESQL and EQUEL. This language is SCAN, described in [ZAHL78]:

```
var ANSWER:packed array [1..3] of char;
begin
   scan SP where P# = 'P4':
      writeln (SP.S#, SP.QTY);
      readln (ANSWER);
      if ANSWER = 'yes' then begin
         SP.QTY := SP.QTY * 1.1;
         modify SP;
      end;
   endscan;
end.
```

fig. 6.1: SCAN-embedded in Pascal.

In this language the basic construct is the scan-loop, processing all rows fulfilling the condition. Within the loop the fields of the current row can be referenced and changed with no restrictions, just like any variable. The actual row is updated when a modify-statement is executed.

The possibility of overriding the prespecifications in a simple way is just as important as the definition of the prespecifications themselves. Otherwise the language will be of no use for application programming, which is a major purpose of embedded languages. That is why we have tried to make an improved embedded language, based upon the proposal from ISO for the embedded database language SQL.This improved language, which we have called Embedded Query Language (EQL), have all the improvements proposed in section 4. A solution to the recurrent problem is shown in fig. 6.2.

```
DCL ANSWER CHAR(3);
CURSOR C SELECT S#, QTY
         FROM SP
         WHERE P# = 'P4';
   PUT SKIP LIST('S# = ',S#,' QTY = ',QTY);
   GET LIST(ANSWER);
   IF ANSWER = 'YES' THEN
         UPDATE CURRENT OF C
            SET QTY = QTY * 1.1;
END-CURSOR;
```

fig. 6.2: EQL-embedded in PL/I.

This example uses a lot of the prespecifications, and consequently it is very short, compared to the embedded SQL program in fig. 2.2, which does exactly the same.

The new program is considerably shorter, since the surroundings, most of the variables and all specification of error-reactions have been removed. The prespecifications will ensure that all error-codes are displayed to the user on the standard output media, and the precompiler will place a standard program heading and footing around the specified statements.

Allmost all constructs in the EQL example are known in the ISO-standard, but one is new, the cursor-loop. It replaces the programmed cursor-loop in fig. 2.2, where the cursor-handling is separated into a declaration, an opening, a fetch and a close of the cursor. In EQL the cursor is declared and opened at the same time, when the loop initiated. Each time the loop is performed, a new row is fetched. When the last row has been processed, the cursor is closed and the program continues after the END-CURSOR statement. The intent has been to declare the semantic contents of the cursor, as close as possible to the actual use of the cursor. Just like in the ISO-standard, the current row of the cursor can be referenced through the CURRENT OF statement, but without having to explicitly declare the cursor "for update".

The CURRENT statement is not the only reason for the explicit naming of the cursor, since it must be possible to have one cursor-loop inside another on the same relation. In order to ensure a unique naming of fields, it must be possible to prefix field-names with the name of the cursor controlling the field.

In order to show how the prespecifications can be overruled by the programmer, fig. 6.3 shows the same program as in 6.2, but with two changes.

First of all the standard transaction handling, which makes the entire program-execution one transaction, has been overruled, in order to make each of the updates a single transaction. This is done simply by stating TRANSACTION COMMIT once for each successfull update.

Second the prespecified error-handling has been overruled, by stating INCLUDE SQLCA in the beginning of the program. This means that all error-handling must be specified explicitly in the program. The inclusion of SQLCA implicitly makes the status-variable SQLCODE available for cheking after the last DB-operation. SQLCODE will be 0 after a successfull operation, and it will be 100 after the last read in a cursor-loop. Note that an error during the reading in a cursor-loop will make the program continue after CURSOR-END.

```
DCL ANSWER CHAR (3);
INCLUDE SQLCA;
CURSOR C SELECT S#, QTY
             FROM SP
             WHERE P# = 'P4';
    PUT SKIP LIST ('S# =', S#, 'QTY =', QTY);
    GET LIST (ANSWER);
    IF ANSWER = 'YES' THEN
    DO;UPDATE CURRENT OF C
            SET QTY = QTY * 1.1;
        IF SQLCODE = 0 THEN
            TRANSACTION COMMIT
        ELSE
        DO; PUT SKIP LIST ('ERROR ON UPDATE');
             PUT SKIP LIST ('ERROR-NO. ',SQLCODE);
        END;
    END;
END-CURSOR;
IF SQLCODE NOT = 100 THEN
DO; PUT SKIP LIST ('ERROR ON SELECTION');
      PUT SKIP LIST ('ERROR-NO. ', SQLCODE);
END;
```

Fig. 6.3 EQL, overriding prespecifications

7. Conclusion

By permitting the use of the constructs from self-contained languages the difference between self-contained and embedded languages is practically removed. This means that there is no longer a need for two languages, since EQL can be just as simple to use as self-contained SQL is today, but it can also be used as a powerful application programming tool by simply specifying a special handling when the standard solutions are not sufficient for the application.

8. REFERENCES.

[ASTR 75] M.M. Astrahan and D.D. Chamberlin: "Implementation of a Structured English Query Language". CAMC, Oct. 1975, p. 580-588

[CHRI86] Anker Christensen: "En sammenligning af interaktive og indlejrede relationsdatabasesprog", DIKU-rapport, 1986.

[DATE84a]: C. J. Date: "A guide to DB2", Addison Wesley 1984.

[DATE 84b]: C. J. Date: "Some Principles of Good Language Design". ACM SIGMOD Record 14 No.3, Nov.1984.

[DATE 84c]: C. J. Date: "A Critique of the SQL Datebase Languages". ACM SIGMOD Record 14 No.3, Nov.1984.

[ISO85] : ISO/TC 97/SC 21/WG 5 - 15: "Database Language SQL", ISO March 1985.

[STON 77]: M. Stonebraker and L. A. Rowe: "Observations on Data Manipulation Languages and their Embedding in General Purpose Programming Languages". Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Memorandum No. UCB/ERL M77/53.

[ZAHL78]: Torben U. Zahle: "SCAN - A simple record at-a-time DML for the relational data model", Proc. ACM-SIGMOD 1978 Int. Conf. on DBMS.