A New Type of Fourth Generation Language for Multimedia Databases: Kasuga Script

Yukari Shirota, Hideaki Nakayama, and Atsushi Iizawa
Software Research Center, RICOH Company, Ltd.
1-1-17, Koishikawa, Bunkyo-ku, Tokyo, 112 JAPAN.
email: {shirota, nakayama, izw}@src.ricoh.co.jp

Abstract
Creating a high-quality multimedia database application is a complex and labor-intensive process. This problem has become especially acute with the advent of contemporary multimedia systems that have become large and powerful at the expense of programming ease and program maintenance. Using traditional relational database fourth generation languages (4GLs), users cannot effectively develop the multimedia database applications of today. Therefore a new type of 4GL is required. In the paper, we introduce a multimedia database application development script called **Kasuga Script**, which our group has developed from scratch. To help users write applications in **Kasuga Script**, **Kasuga Script** offers object classes, both navigational and declarative query functionality, **tags**, **transvalue**, and **operation process**.

Keywords
multimedia database, **Kasuga Script**, **tag**,**operation process**

1 INTRODUCTION

The paper describes a multimedia database application development script called **Kasuga Script**, which our group has developed from scratch. For the design direction of **Kasuga Script**, we adopted a script-oriented approach similar to Tcl/Tk (Ousterhout, 1994) and Perl, which are widely known to increase programming productivity (The USENIX Association, 1995). In addition, as **Kasuga Script** is evaluated by an interpreter, it is also possible to eliminate the delay caused by the edit-compile-execute cycles associated with conventional program development. The system integration (SI) group in our software center has developed many applications using **Kasuga Script**, and found it both easy to use and highly productive. Before we designed **Kasuga Script**, we carried out a detailed investigation of the labor-intensive work involved in multimedia database application programming. This is because we thought that if the critical parts can written easily, the total programming cost would be drastically reduced. We identified the following labor-

N. Terashima et al. (eds.), *Advanced IT Tools*
© Springer Science+Business Media Dordrecht 1996

intensive tasks: (1) making a connection between a GUI widget on a screen and a database field, and a data conversion between them; and (2) writing the many callback procedures. Binding a database field and a GUI widget is one of the biggest jobs in the programming. Another one is writing a lot of callback procedures. Conventional GUI toolkits require the programmer to attach callback procedures to GUI widgets. These procedures are called by the system when the user operates the widget to notify the application of the user's actions. Unfortunately, real interfaces contain hundreds or thousands of widgets, and therefore many callback procedures, most of which perform trivial tasks, resulting in a maintenance nightmare (Myers, 1991).

Kasuga Script offers some mechanisms that allow the user to write these critical parts much more easily. The paper illustrates the features of our **Kasuga Script** and how **Kasuga Script** solves the existing problems of SQL and the traditional relational model (Codd, 1970). The problems are: (1) difficulties with multimedia data independence in the relational model; (2) difficulties with navigational data manipulation in pure SQL queries*; and (3) impedance mismatch problems in embedded SQL.

2 OBJECT-ORIENTED LANGUAGE

For multimedia database application development, Codd's relational model is inadequate because it has no functions for handling multimedia data. As a result, the application has to check for consistency, instead of the DBMS (DataBase Management System). To solve the problem, in recent years, the concept of ADT (abstract data type) or encapsulation in the object-oriented paradigm has been introduced into database programming languages (James, 1983, Osborn and Heaven, 1986). Nowadays, not only object-oriented database programming languages but also 4GLs based on the relational model have imported the solution for multimedia database applications.

Kasuga Script has also been designed as an object-oriented programming language. In **Kasuga Script**, there are a lot of object classes including data manipulation results, visual interface components called **Kasuga widgets**, and image data and a lot of methods are built in it. With multimedia data classes, data in **Kasuga Script** are encapsulated so that data independence can be maintained. In particular, **Kasuga Script** offers variations of the class image data and image handling methods, such as dithering and color map arrangement, and pattern recognition methods, such as OCR and line recognition.

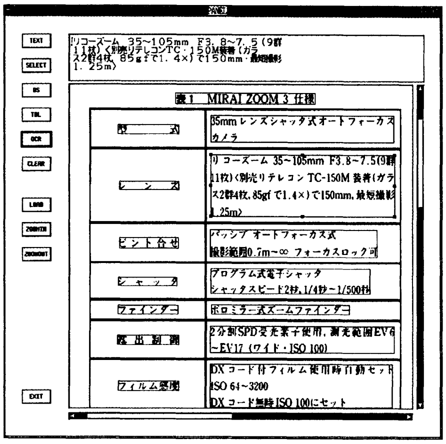
Here is a multimedia application example that illustrates the pattern recognition functions. Suppose that the application utilizes an OCR function to access database field data. The operator need not input data from the keyboard. A **Kasuga Script** program with the OCR functions can be written as shown in Figure 1. Figure 2 shows a screen of the application. Through analyze_table(), table frames and strings in table cells are recognized and then visually framed within borders. When an operator selectes a cell in the table, ocr() is executed for that cell and a string generated by the OCR function is displayed at the top of the screen. The recognition process generates its own internal data called recognition objects. In addition, a Kasuga image widget has a function for displaying a drawing surface where an operator can draw and handle drawing objects, such as lines and rectangles. The two kinds of objects are bound together on the image widget. Hence,

*In the paper, we call a non-extended SQL a pure SQL.

the recognized parts are automatically displayed in a different color or framed in a box just after recognition, and conversely onscreen items or rectangle areas selected by an operator are instantly identified by the recognition process and recognized. Consequently, even if users have no knowledge about the recognition mechanism, they can easily write a multimedia application with advanced pattern recognition functions.

```
var img :image;  /* decralation of variable img */
screen(scl) {    /* create an image widget imw */
 image(imw) :width 500 :height 500;  };
scl.popup();               /* display imw */
img.load("CATALOG image"); /* load image data from a file to img */
scl.imw.put_image(img) :shared true;  /* display the image data */
img.divide_segment();       /* recognize text areas and image areas */
img.analyze_table() :selected; /* analyze tables in img */
 (waiting for an operator toselect a cell in a table)
img.ocr() :selected; /* apply an OCR function to the selected cell */
```

Figure 1: **Kasuga Script** program that uses pattern recognition functions.



リコーズーム 35~105mm F3.8~7.5(9群11枚)〈別売りテレコンTC-150M装着(ガラス2群4枚, 85gfで1.4×)で150mm·最短撮影1.25m〉

表1 MIRAI ZOOM 3 仕様

型　　式	35mmレンズシャッタ式オートフォーカスカメラ
レ　ン　ズ	リコーズーム 35~105mm F3.8~7.5(9群11枚)〈別売りテレコン TC-150M 装着(ガラス2群4枚, 85gf で1.4×)で150mm, 最短撮影1.25m〉
ピント合せ	パッシブ オートフォーカス式 撮影範囲0.7m~∞ フォーカスロック可
シ　ャ　ッ　タ	プログラム式電子シャッタ シャッタスピード2秒, 1/4秒~1/500秒
ファインダー	ポロミラー式ズームファインダー
露　出　制　御	2分割SPD受光素子使用, 測光範囲EV6~EV17 (ワイド·ISO 100)
フィルム感度	DXコード付フィルム使用時自動セット ISO 64~3200 DXコード無時ISO 100にセット

Figure 2: Sample screen of an application that uses pattern recognition functions.

```
path_schema EMP {
   database: /demo/emp;  /* database name */
   path: emp ../belong/.. division as  emp_path;
   var: emp_id   = {emp.ID};
        emp_name = {emp.name};
        div_name = {division.name};
} emp_schema1, emp_schema2;
EMP.open();  /* database open */
emp_schema1.emp_path.generate_path_set(); /* data retrieval operation */
```

Figure 3: **Kasuga Script** program that manipulates data.

3 BOTH NAVIGATIONAL AND DECLARATIVE QUERY

A new type of 4GL such as **Kasuga Script** should also be a full-featured general-purpose programming language. In other words, it must meet computational completeness so that navigational (procedural) procedures can be described using the language. On the other hand, as Stonebraker et al. insist in their "third generation database system manifesto," the declarative query function is essential for data independence and effective data manipulation (Stonebraker et al., 1990). Namely, both navigational and declarative query functions are needed in our database programming language. A pure SQL program does not have navigational functionality. Hence, it has less descriptive power than an object-oriented database programming language. However, nowadays almost all database systems possess their own application development languages or 4GLs, and the majority of users use these languages; the number of pure SQL users is small. These languages provide computational completeness. Therefore, using the application development languages instead of pure SQL, they can solve this descriptive power problem.

Kasuga Script is a general purpose language with computational completeness. Simultaneously, **Kasuga Script** offers declarative query functions in conjunction with RICOHBASE. RICOHBASE is a DBMS that our software division developed. RICOHBASE is based on the Graph Data Model (GDM) (Kunii, 1990) which belongs to the family of extended relational data models. Its main departure from the relational model consists of the introduction of link types, thereby making it a link-oriented model. Figure 3 shows a simple **Kasuga Script** program that manipulates data. The path_schema EMP is defined as a database access path class and emp_schema1 and emp_schema2 are its instances. The path "emp ../belong/.. division" shows a data access path that consists of record type "emp," link type "belong," and record type "division." Variables such as emp_id and emp_name are called **tag**. The above-mentioned emp.ID, emp.name, and division.name represent database field names. Since the three expressions in the var part define **tag** variables corresponding to database fields, after data retrieval operations, we can access the data through the **tags** as follows: emp_schema1.emp_id. In **Kasuga Script**, a method of retrieval is **generate_path_set()** as shown in the example.

4 SEAMLESS LANGUAGE DESIGN

When we write a database query in SQL, an embedded SQL program is often used. However, there is a semantic and syntactical gap between SQL code and its parent language such as C or Fortran. This is called the impedance mismatch problem of SQL. In addition, another gap exists in an application program between control flow parts and the GUI library (see Figure 4.). Because both data structures are different, data conversion is needed, which also gives programmers trouble. The two gaps in embedded SQL programming create labor-intensive work for programmers. To solve the problem, we have designed **Kasuga Script** to be seamless. **Kasuga Script** users, in advance, define database fields to correspond to GUI widgets, using the above-mentioned **tags**. After the definition, data conversion between database fields and GUI widgets is automatically executed, hidden from the operator's view.

Figure 5 illustrates GUI widgets bound with database fields through **tags**. The upper part in the figure illustrates the internal status of the DBMS. Because the DBMS of **Kasuga Script** is RICOHBASE, it shows the data structures of RICOHBASE. In RICOHBASE query processing, the resultant data type is the set of access paths for the query; namely the path consists of record types and their link types. When a query is executed, a **target**, which is a set of paths, is generated together with a cursor pointing to a path. The cursor can be moved up and down over the entire range of the **target** and facilitates record-at-a-time operations. The path pointed to by the cursor is referred to as the current path. The path data structure, as shown in the figure, consists of upper and lower structures. The upper structure is a set of Unique Record Identifiers (URIs) of record occurrences and the lower one is a set of field values. The reason for the two-layered structure is to make the internal data search more effective.

If a **tag** is bound with a database field name, whenever the cursor is moved, the **tag** value also changes so that the **tag** may point to the corresponding field of the current path. In addition, if the **tag** is bound to a widget on the screen, the new value is displayed on the widget. Inversely, when a string is input on the widget, the **tag** value also changes and the corresponding internal field value may change.

In general, complicated data conversion is required between a database field and a widget. Therefore, it is of overriding importance that the data conversion parts be easily written. In **Kasuga Script**, it can be described using the **transvalue** mechanism. The examples shown in Figure 6 illustrate three types of **transvalue** descriptions. In the first example, the **transvalue** is used to encode/decode a nation name and its code numbers. Next, the **transvalue** is used to get a population figure from a database. The input (qualification) data is a nation name and the output is a population figure. The strings "name" and "population" are field names of record "nation." In the last example, input image data is output after scaling in the given dimensions. The scaling factors "w" and "h" are given as parameters. For the action rules, we can write **Kasuga Script** programs. Hence, all **Kasuga Script** methods such as the above-mentioned OCR are available.

5 POST-SPAGHETTI PROGRAMMING

The programming style of GUI toolkits, such as those used in the X window system (McCormack and Asente, 1988), MacApp (Schmucker, 1986), and HyperCard and HyperTalk (Goodman, 1987) is said to be event-driven. There are problems with this programming style: (1) the whole program structure becomes intricate; (2) the logical sequence of operations is separated into multiple procedures; and (3) it is difficult to define some operations as modules, such as data update. In general, we call it the event-driven spaghetti programming problem. For example, suppose that there is a button named $\boxed{\text{EXECUTE}}$ on a screen. The button is supposed to be pushed in multiple contexts such as retrieve, insert, or update. Therefore, the callback procedure invoked when the button is pushed must include various instructions depending on the context:

```
if (mode == RETRIEVE) staff_retrieve();
else if (mode == INSERT) staff_insert();
else if (mode == UPDATE) staff_update();  ........
```

In event-handling programming such as one of GUIs, the modes tend to be nested in a complicated way. Hence the number of conditional branches becomes generally large. To solve the problem, we introduced an **operation process** to allow program moduliza-tion. In **Kasuga Script**, whenever a database operation is invoked, a new **operation process** can be created. The **operation process** internally owns its own stack structure which is separated from those of other **operation processes** and each **operation process** has its own event queue. Figure 7 is a **Kasuga Script** version of the above-mentioned example. As shown here, since there is no need to monitor what is happening in other operations, the program structures become simple and neat. Using wait_event() statements, an **operation process** waits for the event described in wait_event() to be issued. In an application process, plural **event_switches** can be executed in parallel. In an **event_switch** structure, each **operation process** is exclusively executed. In an **operation process** using wait_event(), the control process proceeds sequentially. We can also use an operation process for transactions and error handling.

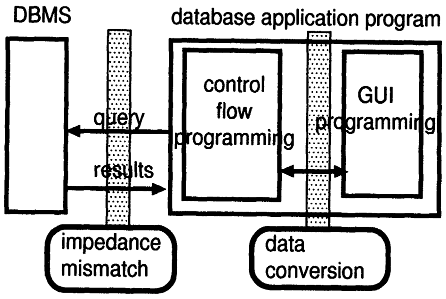


Figure 4 Two kinds of gaps in a database application.

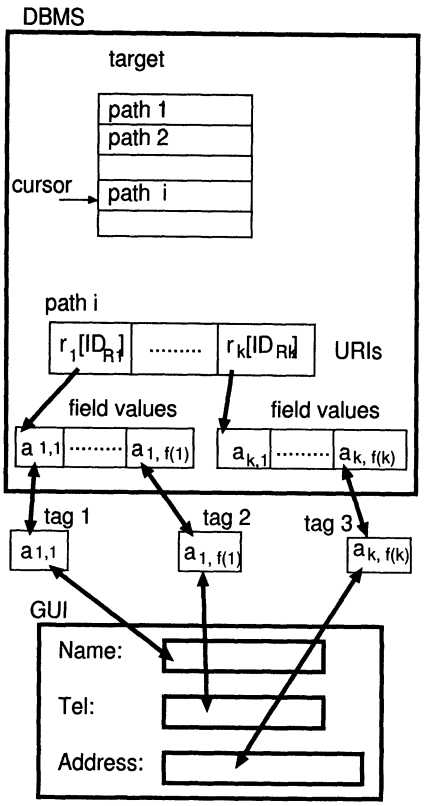


Figure 5 GUI widgets bound to database fields via **tags**.

Let us consider the related work with our **operation process**. The "programming by demonstration" methods in (Myers, 1991), for which the goal is to solve the spaghetti program problem, allows callbacks to be insulated from widgets. However, if the users action sequences are complicated, the users cannot express their required actions only by demonstration. We think that practical, complicated GUI programming still needs textual programs written in languages such as **Kasuga Script**.

The opposite approach to the event-driven style is the state transition machine based style (Jacobs, 1982, Wasserman, 1984, and Olsen, 1984). The problems of the approach are: (1) a user can deal with only one thing at a time and cannot do actions other than in the fixed order; and (2) if a program scale is large, we cannot draw the state transition graph, because the action sequences are complicated and the number of combinations increases exponentially. To solve these problems, the event-driven approach was introduced. The event-driven approach, however, also has some problems as mentioned above. The approach in **Kasuga Script** is in principle oriented to the event-driven style and in addition partially adopts the state transition machine style. The method wait_event() plays the role of a state transition machine and on the other hand widgets are left still available in the event-driven style. By a combination of both approaches, **Kasuga Script** is able to solve the problems of both approaches.

```
/* 1.define a translation table of
 * input and  output data pairs */
transvalue tr_nation(code)(name){
  table: {0, "Italy"},
         {1, "France"},
         {2, "Germany"}
};
/* 2.retrieve a database with
 * input data as the qualification
 * part */
transvalue population(in)(out){
 database: /db/nation;
 path:     nation[name == in];
 value:    out = population;
};
/* 3.write data conversion rules
 * in Kasuga Script */
transvalue
    tr_image(in)(out, w, h){
    action:
     in = out;
     $out.scale() :width $w
                  :height $h;
};
```

Figure 6 Kasuga Script transvalue program examples.

```
event_switch {
 case <active/scr.RETRIEVE>:
 /* button RETRIEVE activated */
  clear_screen();
  /* qualification input by
   * operators until button EXEC
   * is activated */
  wait_event(<activate/scr.EXEC>);
  staff_retrieve();
  break;
 case <active/scr.INSERT>:
  /* button INSERT activated */
  clear_screen();
  /* data input by operators
   * until button EXEC is
   * activated */
  wait_event(<activate/scr.EXEC>);
  staff_retrieve();
  break;     ...........
};
```

Figure 7 Kasuga Script program using an **operation process**.

6 CONCLUSIONS

This paper presents a database programming language named **Kasuga Script**. **Kasuga Script** offers many features: **tags** to bind a database field and a GUI widget; **transvalues** for data conversion; and **operation processes** to avoid event-driven spaghetti programing problems. In database application programming, binding a database field with a widget and converting data between them involve labor-intensive work. In **Kasuga Script**, using **tags**, **transvalue**, and **operation processes**, the size of programs as well as labour costs can be drastically reduced.

REFERENCES

Codd, E.F. (1970) A Relational Model of Data for Large Shared Data Banks. *CACM*, **6**, 377–387.

Goodman, D. (1987) *The Complete HyperCard Handbook.* New york, Bantam Books.

Jacob, R. (1982) Using Formal Specifications in the Design of a Human-Computer Interface. *Proc. of Human Factors in Computer Systems*, March, 315–322.

James, J., Fogg, D. and Stonebraker, M. (1983) Implementation of Data Abstraction in the Relational Database System INGRES. *ACMSIGMOD Record*, April, 1–14.

Kunii, H.S. (1990) *Graph Data Model and Its Data Language*, Springer-Verlag, Tokyo, 1990.

McCormack, J. (1988) An Overview of the X Toolkit. *Proc. of the ACM SIGGRAPH Symposium on User Interface Software*, 46–55.

Myers, B.A. (1991) Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. *Proc. of the ACM SIGGRAPH Symposium on UIST*, 211–220.

Olsen, D. (1984) Push-down Automata for User Interface Management. *ACM Trans. on Graphics*, **3**, 177–203.

Osborn, S.L. and Heaven, T.E. (1986) The Design of a Relational Database System with Abstract Data Types for Domains. *ACM Trans. Database Syst.*, **3**, 357–373.

Ousterhout, J.K. (1994) *Tcl and the Tk Toolkit*, Addison-Wesley.

Schmucker, K.J. (1986) *Object-Oriented Programming for the Macintosh.* Hasbrouck Heights, NJ, Hayden Book Company.

Stonebraker, M. et al. (The Committee for Advanced DBMS Function) (1990) Third-Generation Data Base System Manifesto. *ACM SIGMOD Record*, **3**, 31–44.

The USENIX Association (1995) *Proc. of the Tcl/Tk Workshop*, July 6-8, Toronto, Canada.

Wasserman, A.I. (1984) Developing Interactive Information Systems with the User Software Engineering Methodology. *Proc. of IFIP INTERACT'84*, 611–617.

BIOGRAPHY: Yukari Shirota is Assistant Manager of the Database Research Group at the Ricoh Software Research Center. Her research interests include image database construction tools and GUI generation for database applications. She received her Dsc from the Dept. of Information Science, Faculty of Science, the University of Tokyo. Hideoki Nakayama is also Assistant Manager of the same group. He is responsible for designing and developing RICOHBASE DBMS kernel parts. He has an MSc from the Dept. of Information Science, Faculty of Science, the University of Tokyo. Atsushi Iizawa is the head of the Database Research Group. His interests range from DBMS kernel design to multimedia database applications and image data technology needed for application development. He received his MSc from the Dept. of Information Science, Faculty of Science, the University of Tokyo.