データベースシステム 104−21
（1995. 7. 19）

ビジュアル・プロトタイピングのための製品仕様データベース

角谷 和俊　安武 剛一　田中 裕彦　縄田 毅史　今井 良彦
松下電器産業株式会社 情報通信研究所

製品仕様をオブジェクトモデルで表現することで，製品進化に対応可能な製品仕様データベースを提案する．本データベースでは，製品進化をスキーマ進化としてとらえ，データベースの一貫性を保持するために，動的制約を提案している．また，本データベースでは，従来のテキストベースのデータベースでは扱うことが困難であった，操作仕様などのビジュアルな仕様も管理することが可能である．

我々は，家電製品のソフトウェア開発のために，オブジェクト指向ソフトウェア開発システム *Visual CASE* を開発してる．*Visual CASE* は提案するオブジェクトモデルと製品仕様データベースに基づいて構築されている．本稿では，*Visual CASE* を用いたビジュアル・プロトタイピング手法によって，製品仕様の検討が容易に実現可能であることを示す．また，本システムの実開発への適用についても述べる．

A Product Specification Database for Visual Prototyping

Kazutoshi Sumiya　Kouichi Yasutake　Hirohiko Tanaka
Takeshi Nawata　Yoshihiko Imai

Matsushita Electric Industrial Co., Ltd.
Information and Communications Technology Laboratory
1006, Kadoma, Kadoma-Shi, Osaka, 571 JAPAN
E-mail:{sumiya, yasutake, hirohiko, nawata, imai}@isl.mei.co.jp

We propose a product specification database which is suited to product evolution, modeling the product specification as an object. In this database, we propose a behavioral constraint to maintain consistency. Furthermore, this database can manage visual specification, such as operational specification, which is hard to handle in an ordinary database.

We have been developing *Visual CASE*: an object-oriented software development system for home appliances. *Visual CASE* is based on the object model we propose. In this paper, we show that the product specification is easy to examine, using visual prototyping. We also discuss implementation issues of the database applied to the home appliance software development process.

1 Introduction

Prototyping methodologies have been of great interest recently, and many results have been presented. However, most of these approaches are applicable to programs but not to other specifications, such as user operations[1]. The user operations are the most important factors, especially in the area of products with SUI (solid user interface), for example, control machines and home appliances. It is very difficult to design a specification of the product, because the specification is too complex to describe on text and on paper documents. To solve the problem, we have been developing *Visual CASE*: an object-oriented software development system[15][2]. This system is based on the object model we propose. The idea of the object model is to incorporate the container object model[6] with the constraints on the message passing mechanism and inheritance scheme.

Meanwhile, many new models of equipment such as microwave ovens and washing machines are put on the market at least annually. Home appliances are characterized by the constant releasing of newly designed products day after day. There are many models and many designs for one piece of equipment. For example, for microwave oven - `economy-model`, `grill-model`, and `convection-model` are models with differing functions. `English-design`, `French-design`, and `German-design` are designs for specific markets. Generally, there are many candidates for specification in real manufacture management. In our experience, 100s of candidates must be examined to produce one product. As the divisions produce 100s of products annually for just one change in basic model specifications, 10000s of candidates must be examined.

Candidates are regarded as versions of the products. The version graph of a product family is very complex because there are many versions in a certain basic model and the basic model evolves itself frequently. Several version models and configuration management techniques have been proposed[4][13]. However, most of these models and techniques are not efficient at maintaining consistency among versions in large quantities. On the other hand, multi-media database systems provide the framework to handle many kinds of data[7]. However, these systems can not handle the specifications, such as user operation and indication of blinking LEDs and lamps.

Our approach to solving these problems is to make clear the relationship between a new basic model and an old basic model. This is in respect to schema maintaining. The class libraries are designed as candidates of the components, and sets of the instance objects are designed as product specifications. Our goal is to provide the objects with high flexibility and reusability of product specifications. The flexibility of the objects enables product designers to modify the product specifications partially in a rapid and intuitive way. In other words, they can prototype the product specifications in a trial-and-error manner. The reusability of the objects makes it easy to keep track of product evolution. It enables product designers to review the past specifications which correspond to the up-to-date specifications. Consistency needs to be guaranteed between the class hierarchies and the instance objects when the class hierarchy is being evolved. We developed a database system to manage the class library and the instance objects, using a release method.

The remainder of this paper is divided as follows: Section 2 discusses requirements for visual prototyping. Section 3 gives the data model and version management. Section 4 discusses implementation issues of *Visual CASE*. Section 5 summarizes our results and suggests our future plan.

2 Visual Prototyping

Prototyping is effective in enhancing design quality in the product development process, especially in the software development process. Developers can examine many candidates for a product through a trial-and-error method. Many prototyping methodologies have been proposed[1]. Most of these are designed for software development. However, it is also necessary for developers to manage other kinds of specifications. We propose a prototyping methodology which is applicable to these specifications, using visual description.

We discuss properties satisfied in a prototyping system. In [3], the software prototyping environment should satisfy the following properties: (1) Executability, (2) Fitness to target environment, (3) Rapid constructibility and modifiability, (4) Refinability in stepwise fashion. In property (1) and (2), an executable language and environment should be satisfied. In property (3) and (4), a data management method should be established.

We claim that two properties are required in the product manufacturing process: **visualization** and **reusability**. The four properties described above are certainly important, however, these two properties are highly effective for rapid prototyping.

- **Visualization:**
  Program and specifications should be illustrated to users. In addition, a visual interface should be provided to construct the specifications.
- **Reusability:**
  Soundness of version graph should be maintained in the development process. The old components should work in the current schema.

Several visual prototyping methodologies have been proposed[14]. One other prototyping tool for machine control interfaces is CISP[5], which is an extension of Apple's HyperCard, offering a series of features built on top

of the standard HyperCard capabilities. This tool allows the user to simulate a system interface by clicking buttons on the CRT display. CISP is applied to the interface design of VCRs. In this tool, there are two problems as follows: One is that the design discussed cannot be handled in the target system directly. The other is that the approach could become unwieldy if care is not taken during the scaling-up process, though it is easy to handle on a small scale.

3 Product Specification Database

In order to realize the visual prototyping of home appliances, we propose the construction of a prototyping system based on a database system storing product specifications. This database system is the first of its kind. In other words, this database system is a specially designed multi-media database for home appliance development. We call this database system a **product specification database**. In this section, we propose the software model that represents product specifications, and the version management of the database.

3.1 Object Model

For the software model for home appliances, we claim that a product specification is represented by functions and user operations to fire them. To represent product specifications, we apply our idea to the object-oriented approach[12]. In other words, we view each product specification as an object: a **product specification object**. In addition, a product specification object contains other objects: **component objects**.

3.1.1 Product Specification Object

A product specification object is a container object whose constituent elements are some component objects. A product specification object corresponds to one particular product in the real world. A component object represents its function. Examples of component objects in a washing machine are power button, timer, water level LED, washing cycle button, and washing cycle.

The set of component objects is structured as a class hierarchy (i.e. class library): a **component class hierarchy**. In this class hierarchy, a descendant class inherits from ancestral class information. Figure 1 shows a product specification object that contains several component objects. The arrows between objects indicate the messages.

It is possible to compose several product specification objects from one component class hierarchy. In general, a container object captures the framework to include its content objects and the operational mechanism to constrain them. Further discussions about container objects can be seen in [17]. Unless the container object offers any constraint, its constituent elements are free to enter and leave their container. Therefore, container objects can offer a rather more flexible environment than the one that composite objects provide since product designers are allowed to attach and detach constituent elements to the product specification.

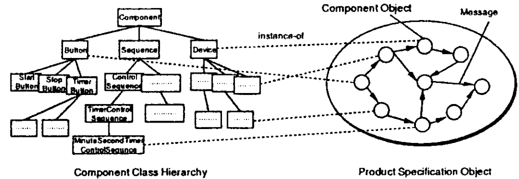

Figure 1: Component Class Hierarchy and Product Specification Object

LAUSIV[1] is a programming language in which the object model we described is implemented. It is like well-known object-oriented languages such as C++ and Objective-C. The inheritance scheme of the state attribute is extended in this language because the state attribute must be considered distinct from other general attributes. In addition, constraints among classes on the extended messages passing mechanism are adopted.

```
class TimerControlSequence : ControlSequence{
 /* definition of state attributes */
 state:
   timer_state =
          {'waiting','setting','executing'};
                ....
                ....

 /* definition of general attributes */
 attribute:
   integer    start_time;
   integer    end_time;
   integer    interval;
                ....
                ....

 /* definition of behavior */
 behavior:
   SetTimer  from < class TimerButton> {
        if (timer_state == 'waiting'){
             timer_state = 'setting';
             interval = end_time - start_time;
        }
                ....
                ....
   }
}
```

The above example describes the component class `TimerControlSequence`, which is a direct descendant of `ControlSequence`.

[1]There is no meaning, but it is simply the word "visual" reversed.

3.1.2 Consistency Management

Several frameworks for schema updates have been proposed [11] [18]. In [18], two basic types of consistency are discussed, namely *structural* and *behavioral* consistency. Structural consistency refers to the static characteristic of the database, and behavioral consistency refers to the dynamic part of the database. The behavioral consistency is too severe to maintain schema, however, it is certainly useful to check class hierarchies. Especially when a schema evolves frequently (i.e. prototyping), we consider that the consistency should allow a certain behavioral *inconsistency*. We introduce **weakly behavioral consistency** to maintain schema reasonably. Weakly behavioral consistency is maintained by the two types of constraint given below. The constraint prevents the method from failing (i.e. run-time errors) and from changing the behavior (i.e. the expected method's result is different).

In the constraint we propose, a component object can designate a component class as the receiver class instead of a particular instance of the class in sending a message. The message issued by the object will be delivered to the object(s) belonging to the receiver class if such object(s) exists in the container object. Otherwise, the constraint mechanism will look for another object that belongs to the descendant of the designated receiver class. If no such objects are found, the message will be ignored or cause an error reply as in the former case. Also, a component object can designate a component class as the sender class for a particular behavior. Namely, the behavior will be fired only by the messages that the objects belonging to the sender class or its descendant classes dispatch. Messages sent from unspecified classes will be discarded or cause an error reply. As a whole, our proposing constraint is characterized by the following:

Sender Constraint: the message sender can specify a receiver class instead of a particular object in sending messages. The sender constraint is represented by the following notation:

```
<class ReceiverClassName> <- MessageName
```

Receiver Constraint: the message receiver can specify a sender class in declaring behavior. The receiver constraint is represented by the following notation:

```
MessageName from <class SenderClassName>
```

In the following example, we show the constraints in Figure 2.

```
/* Sender Class */
class TimerButton : Button {
                   ....
 behavior:
  ButtonOn {
                   ....
    [< class TimerControlSequence> <- SetTimer];
                   ::::
  }
}
```

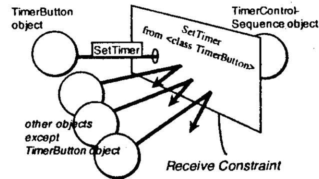

Figure 2: A Constraint among Component Objects

```
/* Receiver Class */
class TimerControlSequence : ControlSequence {
                        ....
 behavior:
  SetTimer  from < class TimerButton> {
                        ::::
  }
}
```

In Figure 2, the component class `TimerButton` declares the component class `TimerControlSequence` as a receiver class of the message `SetTimer`. Similarly, the component class `TimerControlSequence` designates `TimerButton` as a sender class of `SetTimer`. As a result, the relationship between `TimerButton` and `TimerControlSequence` is described by the constraint imposed on the message sending mechanism relating to `SetTimer`.

3.2 Version Management

There are many versions of a product specification object, because it is possible to compose several product specification objects from one component class hierarchy. For example, for a microwave oven – `'94-English-design`, `'94-French-design` and `'94-German-design` are composed from the component class hierarchy `'94-GRILL-MODEL`. On the other hand, a component class hierarchy is evolved by adding classes, modifying classes, and removing classes. For example, a product modification for a microwave oven – from `'94-GRILL-MODEL` to `'95-GRILL-MODEL`, the component class `10-MinutesButton` is attached and the component class `SteamSensor` is modified. The relationship between the component class hierarchy and the product specifications may be contradictory in the evolution. For example, as `SteamSensor` is modified in the product modification, `'94-English-design` and `'94-German-design` will work. However, `'94-French-design` won't work, because the combination of new `SteamSensor` and `'94-GRILL-MODEL` components are not compatible *only* in this case.

We propose a configuration management method to solve this problem, which is called the **release method**. This method prevents a component class hierarchy de-

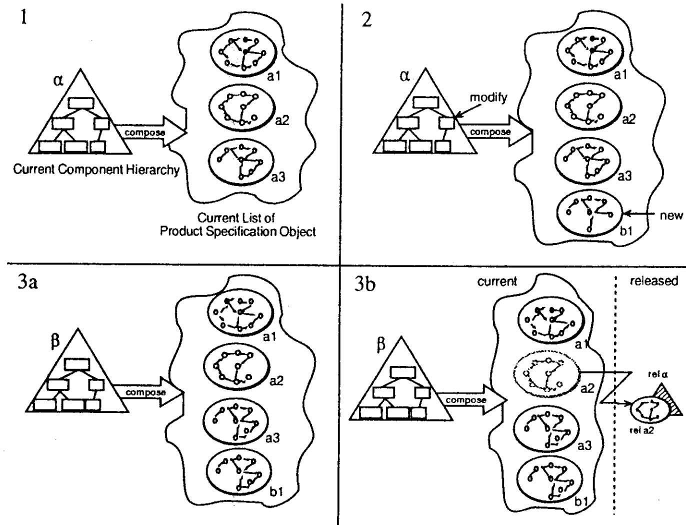

Figure 3: Release Method

structing if its hierarchy evolves. Figure 3 shows the release method as follows:

Phase 1 The product specification object **a1** is composed from current component class hierarchy α. In the same way, **a2** and **a3** are also composed. In this case, the current list of product specification objects includes **a1**, **a2**, and **a3**.

Phase 2 A class in α is modified and new product specification object **b1** is composed. At this time, if **a2** has a modified class object, we must check whether the product specification is contradictory to α. If it is not contradictory, go to Phase 3a. Otherwise go to Phase 3b.

Phase 3a The current component class is β evolved from α and the current list of product specification objects includes **a2**.

Phase 3b The current component class is β evolved from α and **a2** is released to *rel a2* with *rel* α. In this case, the current list of product specification objects does'nt include **a2**.

The released version of the product specification object is detached from the current list of product specification objects. At this time, the component class hierarchy, from which the product specification object is composed, is detached and stored with the product specification object. The reason why the component class hierarchy is also stored is as follows: (1) The product specification object is guaranteed to work completely. (2) The component class hierarchy evolves individually. In this way, it is easy to distinguish the released version from the current main version of the component class hierarchy. For example, the released version of the component class hierarchy `NorthEuroean-MODEL` evolves to `Sweden-MODEL` and `Norway-MODEL`, and still more branches to `NorthAmerican-MODEL` and so on.

We implement our object model on two databases. One of the databases is the **product specification database** which manages versions of product specification objects. The other database is the **component database** which manages the versions of class hierarchies. We compose product specification objects in the product specification database from component objects defined in the component database. Figure 4 shows the relationship between the component database and the product specification database.

4 Implementation

In this section, we describe implementation issues of the database functions in the manufacturing process. In section 4.1 we describe the system architecture of the *Visual CASE* system. In section 4.2, we discuss evaluation of *Visual CASE* system.

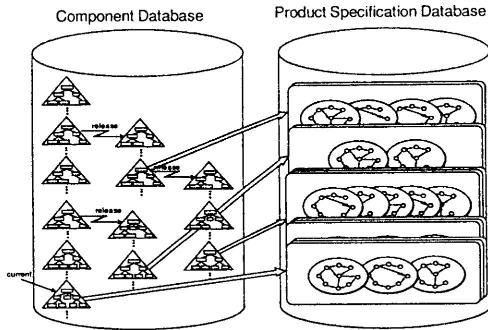

Figure 4: Component Database and Product Specification Database

4.1 Architecture of Visual CASE

Visual CASE is a software development system specifically designed for the embedded software in home appliances and provides a framework which can be used by all the developers: product planners, product designers, and software developers. The architecture of *Visual CASE* supports various software development stages from the conceptual specification design to executable code generation. *Visual CASE* runs on Sun OS with Open Windows 2.0 and Object-Oriented Application Development Software "ActivePage" [8].

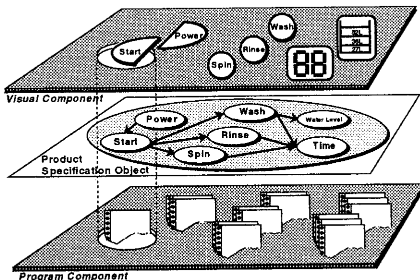

Figure 5: Visual Components and Program Components

The component objects dealt with in *Visual CASE* have not only a level representing a function of a component but another two levels. In other words, a component object is linked to two subcomponents: a visual component and a program component. To simulate product operations, *Visual CASE* uses the visual component. To synthesize the executable program, *Visual CASE* uses the program component. Figure 5 shows the relationship of components and these subcomponents.

Figure 6 shows the architecture of *Visual CASE*. *Visual CASE* consists of six tools and five managers[2]. The tools provide the developers with the interface to manipulate products and components in the product specification database and the component database. The managers provide the tools with the interface to access the product specification database and the component database.

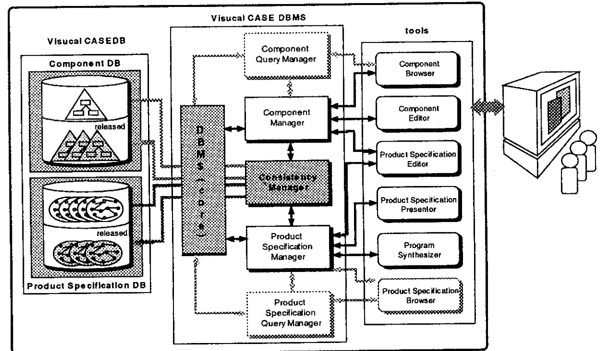

Figure 6: Architecture of Visual CASE

The **component editor** provides the developers with an interface to create, delete and modify a component object. The **component browser** provides the developers with an interface to traverse a component class hierarchy and paste a component object on a product specification object. The **product specification editor** allows the developers to create, delete and modify a product specification object. The **product specification presenter** allows the presentation of the appearance of a product specification object on the CRT display. The developers can operate the 'pseudo' product on the CRT display. The **program synthesizer** generates a control skeleton of the program. This synthesizer uses program components to collect program fragments. The **product specification browser** provides the developers with an interface to traverse the product specification database. These tools have a graphical user interface on the CRT display.

The **component manager** receives the request to retrieve and store the component objects from the component browser and the component editor, and to pass the class definitions to the product specification editor. The

[2]The product specification browser, the component query manager, and the product specification query manager are yet to be implemented.

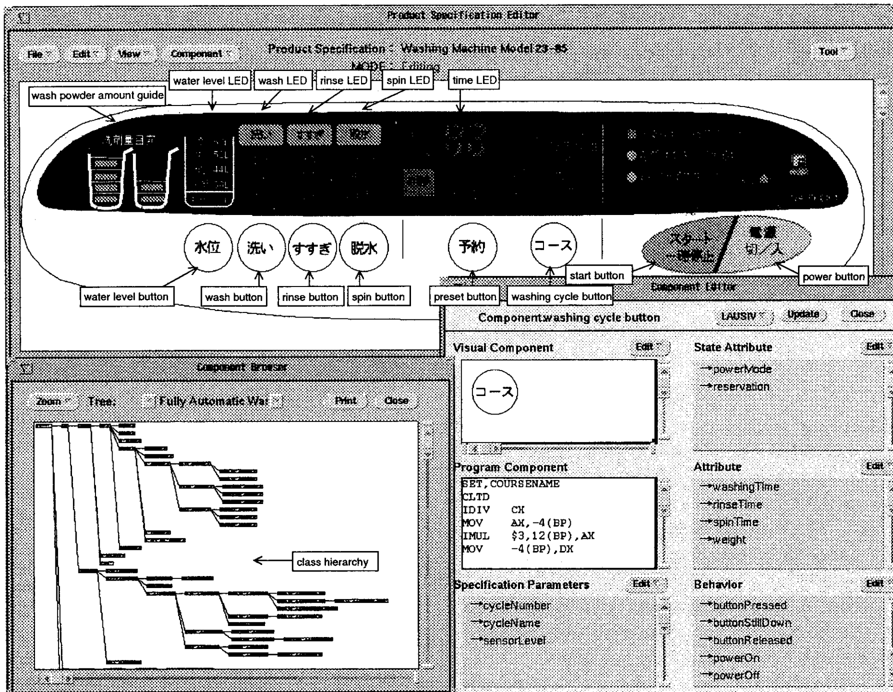

Figure 7: Screen Image of Development using Visual CASE

product specification manager receives the request to retrieve and store the product specification objects from the product specification editor, product specification presenter, and program synthesizer. The **component query manager** and the **product specification manager** receive the request to search the component object and the product specification object from the component browser and product specification browser.

The **consistency manager** observes the consistency between the current class hierarchy and the current list of product specification objects. The component manager receives the request from the cnsistency manager to check which class hierarchy is current and which classes are modifying. The product specification manager receives the request from the consistency manager to check the current list of product specification objects. The release manager transfers the released component class hierarchy and product specification objects into the component database and the product specification database respectively, using the access methods of the DBMS(core).

The *Visual CASE* DBMS(core) provides access methods of the component database and the product specification database to all managers. The DBMS is implemented on ActivePage. As ActivePage adopts Objectivity/DB[10] as a storage manager, the DBMS indirectly accesses Objectivity/DB through ActivePage standard functions. The component database has two storages: working storage and released storage. The product specification database has also two storages: working storage and release storage. The working storages include the current versions and the released storages include the released versions.

4.2 Evaluation

Figure 7 shows the screen image usign *Visual CASE* to examine the specifications of a washing machine[3][9]. In Figure 7, the bottom right part of shows the product specification presenter presents all the visual subcomponents of the component objects contained in the product specification object of a particular washing machine. The bottom right part shows the view of the component browser for a particular component object. The top part of shows the

[3]The control panel of this product was designed by *Visual CASE* and actually put on the market.

view of the component editor.

Our study shows that *Visual CASE* reduced the time to fix the initial conceptual design by a factor of 20%[16]. The major reason for this remarkable effectiveness is the fact that *Visual CASE* eliminates the unnecessary productions of physical mockups thanks to its visual prototyping ability. Due to the interdependent relationship between the parts of the development process, if the design of the part could not be decided, the next stages would also become delayed. As a whole, *Visual CASE* can cut 50% off the time of overall software production processes.

5 Conclusions

We have described a framework for prototype techniques of software development. Our approach is to design a data model for product specifications: the product specification object and the component object, to provide the release method and to construct a product specification database. The main advantage of the database is its ability to manage the consistency of class hierarchies and instance objects in large quantities.

We have also discussed implementation issues of the database applied to *Visual CASE*: an object-oriented software development system for home appliances. *Visual CASE* has been applied to the real manufacture management process. A control panel designed by *Visual CASE* has actually been put on the market. The case study has shown that *Visual CASE* reduced the time to fix the initial conceptual design effectively and the users continuously made good use of *Visual CASE* for the development process.

The four properties described in Section 2 are satisfied in *Visual CASE* as follows: (1) The system can examine functions and performance using visual description. (2) The system can maintain compatibility between prototype and target software by synthesizing a program from the code fragments. (3) The system can easily modify the specification with an interface through visual tools using the databases. (4) The system can manage the evolution of the specifications by the release method while maintaining consistency. We also have a plan to verify and test the specifications on this object model.

Acknowledgments

We gratefully acknowledge helpful discussions with Yoshifumi Masunaga, professor at University of Library and Information Science, on several points in this paper. We would also like to thank Katsumi Tanaka, professor at Kobe University, for the advice on the model we propose. *Visual CASE* is a result of a team effort. Other team members include Norio Sanada, Takuya Sekiguchi, Toshihiro Hishida, and Satoshi Kawabata.

References

[1] V. Scott Gordon and James M. Bieman. Rapid prototyping: Lessons learned. *IEEE Software*, 12(1):85–95, 1995.

[2] Y. Imai, K. Sumiya, K. Yasutake, and S. Haruna. Visual CASE: A Software Development System for Home Appliances. *COMPSAC93*, pages 11–18, 1993.

[3] K. Itoh, Y. Tamura, and S. Honiden. TransObj: Software prototyping environment for real-time transaction-based software system applications. *Software Engineering and Knowledge Engineering*, 2(1):5–30, 1992.

[4] R. H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, 1990.

[5] Halskov Kim and H. Peter Aiken. Experiences Using Cooperative Interactive Storyboard Prototyping. *CACM*, 36(4):57–64, 1993.

[6] W. Kim, J. Banerjee, H. T. Chou, and J. F. Garza. Composite Object Revisited. *ACM SIGMOD*, pages 337–347, 1989.

[7] Y. Masunaga. Design issues of OMEGA: An object-oriented multimedia database management system. *Journal of IPSJ*, 14(1):60–74, 1991.

[8] Yoshiyuki Miyabe. Object-Oriented Multi-Media Application Development Software. *8th German-Japanese Forum on Information Technology*, 1993.

[9] Y. Nukina, W. Uchiyama, H. Fujii, Y. Omura, K. Iwamoto, and H. Tanaka. Washing machine with double cascades. *National Technical Report*, pages 3–9. Matsushita Electric Industrial Co., 1995. (in Japanese).

[10] Objectivity. *Objectivity Database System Overview*. Objectivity Inc., 1990.

[11] Sylvia L. Osborn. The Role of Polymorphism in Schema Evolution in an Object-Oriented Database. *IEEE on Trans.Knowledge and Data Engineerings*, 1(3):310–317, 1989.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[13] E. Sciore. Multidimensional Versioning for Object-Oriented Databases. In *proceedings of 2nd DOOD*, 1991.

[14] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold, 1991.

[15] K. Sumiya, T. Ohtsu, S. Haruna, and Y. Imai. Visual CASE: An Object-Oriented Software Development System for Home Appliances. *TOOLS USA*, pages 97–107. Interactive Software Engineering, 1993.

[16] H. Tanaka, S. Abe, W. Uchiyama, E. Ishizaki, T. Nawata, and Y. Imai. Prototyping System for Home Appliances – Case Studies in Control Panel Design. *Quality Management Symposium on Software Production*, pages 9–16, 1994. (in Japanese).

[17] K. Tanaka, S. Nishio, M. Yoshikawa, S. Shimojo, and T. Jozen. Obase: An Instance-Based Object Database System with Dynamic Inheritance and Active Rule Mechanisms. *IPSJ SIGDBS Tech. Rep.*, 94-DBS-100, pages 87–96, 1994.

[18] R. Zicari. A Framework for Schema Updates In An Object-Oriented Database System. *ICDE*, pages 2–13, 1991.