# A Product Specification Database for Visual Prototyping

Kazutoshi SUMIYA
*sumiya@isl.mei.co.jp*

Kouichi YASUTAKE
*yasutake@isl.mei.co.jp*

Hirohiko TANAKA
*hirohiko@isl.mei.co.jp*

Norio SANADA
*sanada@isl.mei.co.jp*

Yoshihiko IMAI
*imai@isl.mei.co.jp*

Matsushita Electric Industrial Co., Ltd.
Information and Communications Technology Laboratory
1006 Kadoma, Kadoma-shi, Osaka 571, JAPAN

## Abstract

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 a visual prototyping system 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.



**Proceedings of the 21st VLDB Conference**
**Zürich, Switzerland, 1995**

## 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[GB95]. 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[SOHI93][ISYH93]. This system is a visual prototyping system based on the object model we propose. The idea of the object model is to incorporate the container object model[KBCG89] 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[Kat90][Sci91]. 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[Mas91]. 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. Generally consistency needs to be guaranteed between the class hierarchies and the instance objects when the class hierarchy is being evolved[Osb89] [Zic91]. 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 and *Visual CASE*. Section 3 gives the data model, the version management, and the query of product specification database. Section 4 discusses implementation issues of *Visual CASE*. Section 5 summarizes our results and suggests our future plan.

# 2 Visual Prototyping

Several prototyping system have been proposed. In Section 2.1, we discuss essential factors of software prototyping. In Section 2.2 we describe *Visual CASE* and evaluate this system.

## 2.1 Prototyping System

We discuss properties which should be satisfied in a prototyping system. In [ITH92], the software prototyping environment should satisfy the following properties:

1. Executability
2. Fitness for 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.

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[GB95]. Most of these are designed for software development. However, it is also necessary for developers to manage other kinds of specifications.

Several visual prototyping methodologies have been proposed[Shu91]. One other prototyping tool for machine control interfaces is CISP[KA93], 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.

## 2.2 Visual CASE

We claim that visualization is required in the product manufacturing process because program and specifications should be illustrated to the designers. In addition, a visual interface should be provided to construct the specifications. We have been developing *Visual CASE*: an object-oriented software development system for home appliances and released the *Visual CASE* system to several divisions where home appliances are produced. These divisions have been applying the system to case studies of their software manufacture management.

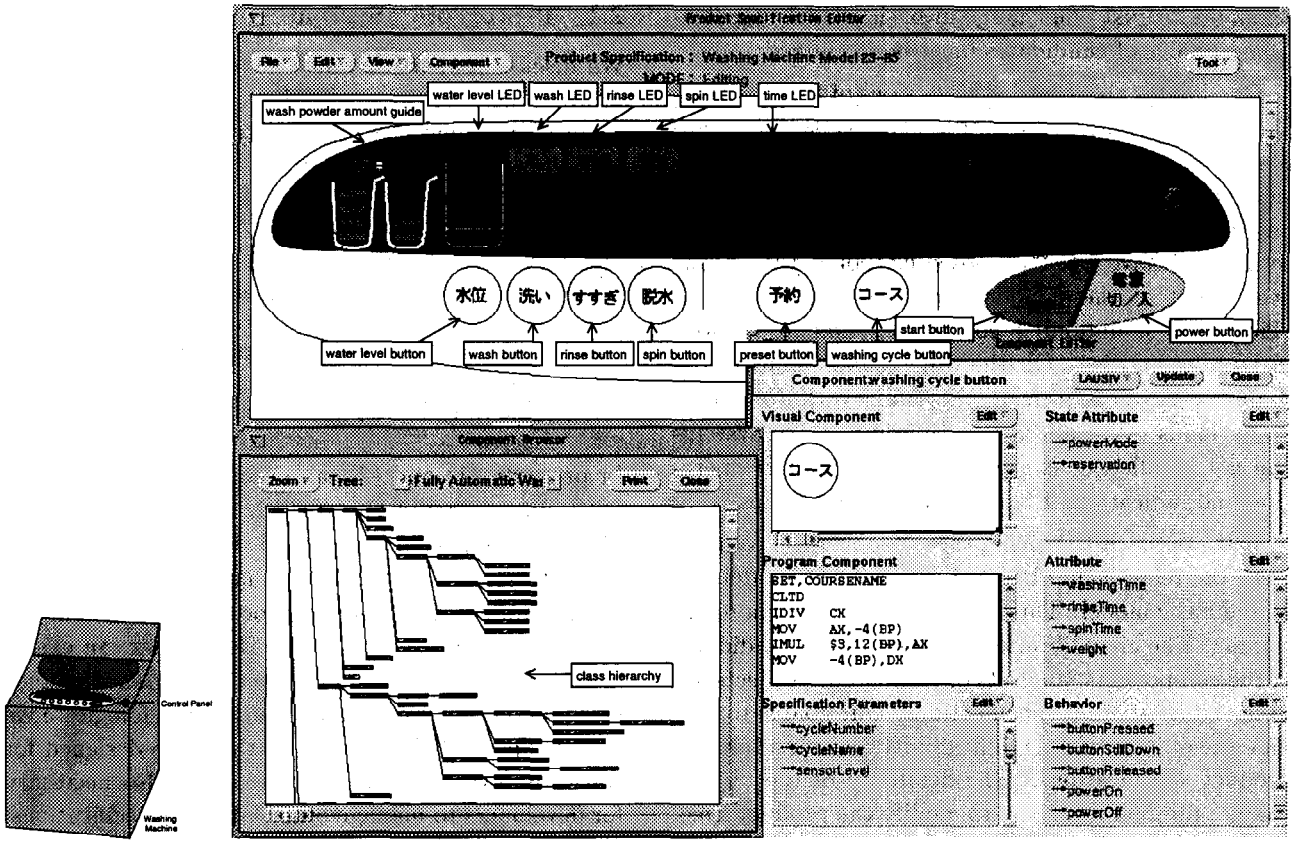


Figure 1: Screen Image of Development using Visual CASE

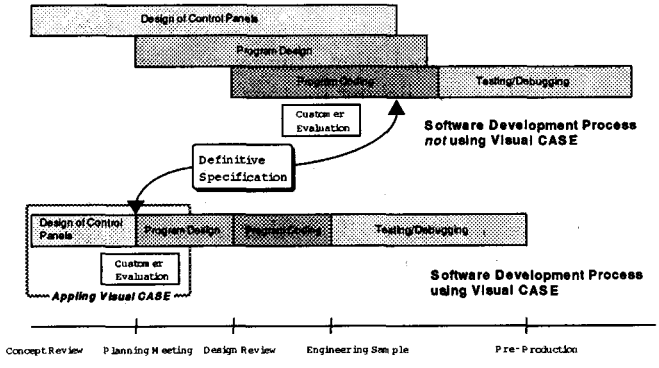


Figure 2: Comparison of Development Processes

Figure 1 shows the screen image using *Visual CASE* to examine the specifications of a washing machine[1] [NUF+95]. With the view shown in the figure, the product designers can operate the 'pseudo' control panel of a washing machine on the CRT display by touching various displayed buttons. The control panel on the CRT display will behave as if it were the actual physical product; the displayed remaining time will be decremented and the wash LED will start blinking if the start button is activated by clicking it on the CRT display.

Figure 2 shows the comparison of development processes: software development process *not* using *Visual CASE* and software development process using *Visual CASE*. The study shows that *Visual CASE* reduced the time to fix the initial conceptual design by a factor of 20%[TAU+94]. 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 components of the development process, if the design of the component 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.

We have applied *Visual CASE* in several divisions and was effective, for *only* one individual basic model. However we must manage the version of candidates in *Visual CASE*, when many basic models are designed.

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

Soundness of version graph should be maintained in the development process. The old components should work in the current schema. Therefore we designed a product specification database for *Visual CASE*. *Visual CASE DB* is a product specification database for home appliances. In Section 3 we describe the desired features for the object mode, version management, and queries.

# 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, the version management of the database, and the query using operation-sequence.

## 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[RBP+91]. 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 3 shows a product specification object that contains several component objects. The arrows between objects indicate the messages. There is no relationship (i.e. part-of) among contained component objects in a product specification object.

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 [TNY+93]. 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.

*LAUSIV* is a programming language in which the object model we propose is implemented[2]. 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.

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

```
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;
        }
               ....
               ....
   }
}
```

Each class has three parts, which are state attribute, general attribute and behavior. In this example, three

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

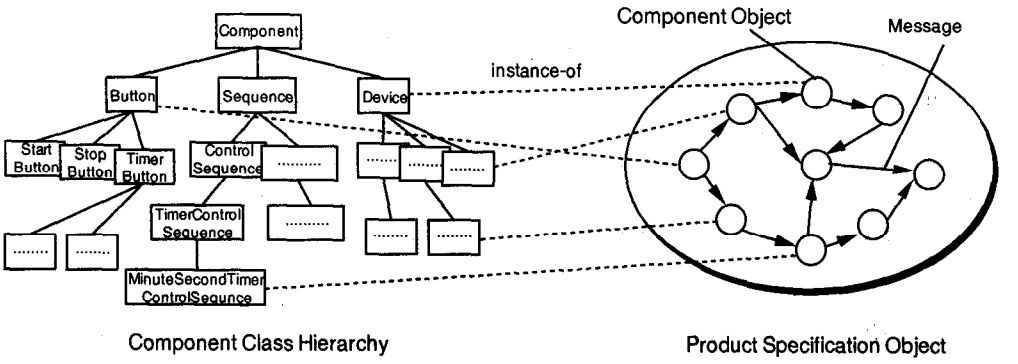


Figure 3: Component Class Hierarchy and Product Specification Object

states are assigned to the state attribute `timer_state`. Three integer variables are declared in the general attributes. In the behavior `SetTimer`, it is declared that the message `SetTimer` is received only from a class which is a descendant of class `TimerButton` and the procedure is carried out when the message is received.

In the example shown below, the component class `MinuteSecondTimerControlSequence`is a descendant class of the component class `TimerControlSequence`. `TimerControlSequence` which has a state attribute `timer_state`, which is assigned to either state `waiting`, `setting`, or `executing`.

```
class TimerControlSequence {
 state:
  timer_state =
       {'waiting, 'setting', 'executing'}
}

class MinuteSecondTimterControlSequence :
                    TimerControlSequence {
 state:
  timer_state.setting =
     {'setting_minute', 'setting_second'}
}
```

In `MinuteSecondTimerControlSequence`, the state `setting` is refined to `setting_minute` and `setting_second`. In this example, `setting` is a generalized state for `TimerControlSequence`, while `setting_minute` and `setting_second` are refined states for `MinuteSecondTimerControlSequence`.

### 3.1.2 Consistency Management

Several frameworks for schema updates have been proposed [Osb89] [Zic91]. In [Zic91], 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 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. 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 4.

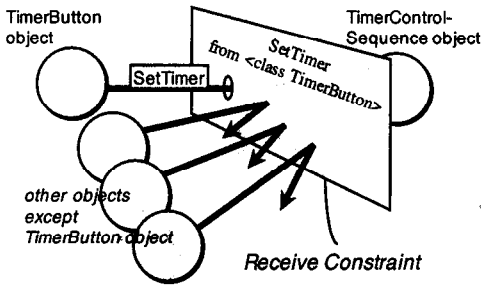


Figure 4: A Constraint among Component Objects

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

In Figure 4, 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 destructing if its hierarchy evolves. Figure 5 shows the release method as follows:

**Phase 1** The product specification object `a1` is composed from current component class hierarchy $\alpha$. 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 $\alpha$ 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 $\alpha$. If it is not contradictory, go to Phase 3a. Otherwise go to Phase 3b.

**Phase 3a** The current component class is $\beta$ evolved from $\alpha$ and the current list of product specification objects includes `a2`.

**Phase 3b** The current component class is $\beta$ evolved from $\alpha$ and `a2` is released to *rel a2* with *rel* $\alpha$. In this case, the current list of product specification objects doesn't 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

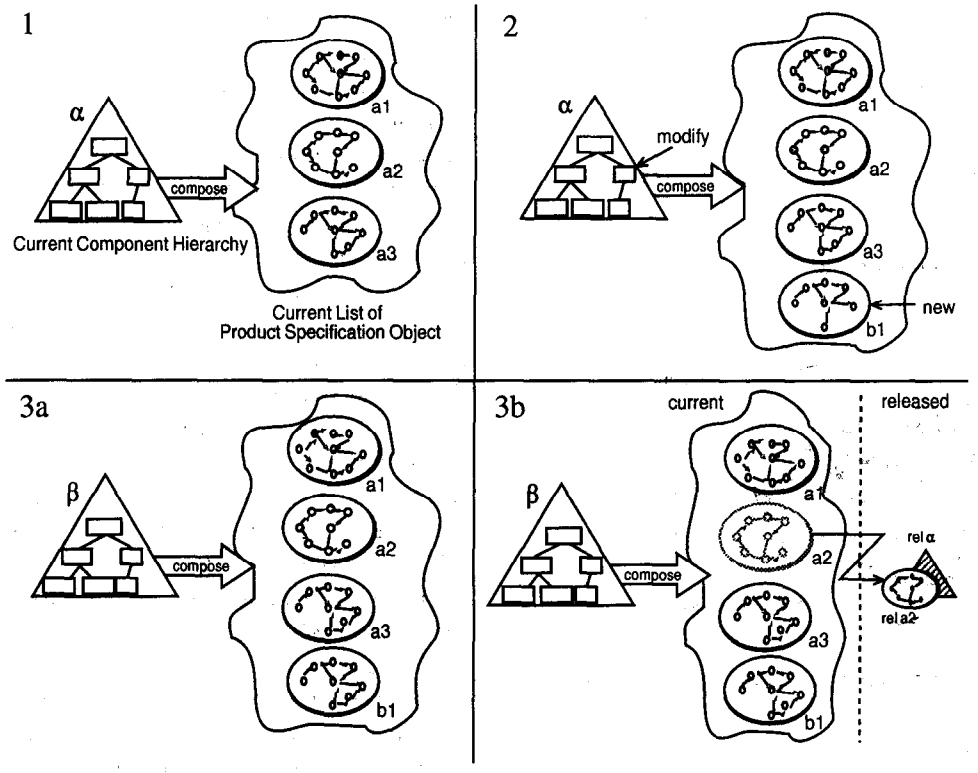


Figure 5: Release Method

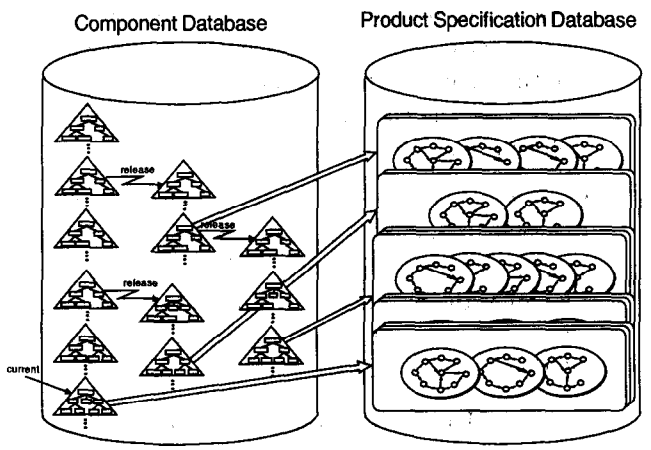


Figure 6: Component Database and Product Specification Database

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 `NorthEuropean-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 6 shows the relationship between the component database and the product specification database.

### 3.3 Query by Operation-Sequence

We claim a new function for visual prototyping should be provided. We propose a query using operation-sequence[SYS+92][3]. The mechanism uses an operation-sequence as a query though the conventional database uses textual language (like SQL).

When an operation-sequence is carried out, a state or states within one or more components within a product changes. We have designed an extension to the conventional class object, the state attribute. A

[3]The query mechanism in this section is still in the planning stage, so it is still to be implemented.

specifically designed inheritance mechanism allows the abstraction and refinement of states. A state is maintained within a component so that state can therefore be abstracted to allow the comparison of states within different components.

There are many kinds of operation-sequences for different operating equipment. For example, for setting the timer of a VCR;

*product* [*button1, button2,...*]

**A** [Timer, Month, Day, Hour, Minute]

**B** [Timer, TapeMode, Hour, Minute, Month, Day]

**C** [Timer, Hour, Minute,TapeMode]

**Example 1:** Find the operation-sequence of product A, corresponding with operation-sequence [Timer,Hour,Minute].

**get** [Timer, Hour, Minute] of {A}

The query of Example 1 returns the operation-sequence:

[Timer, Month, Day, Hour, Minute]

**Example 2:** Find the most similar product to operation-sequence [Timer, TapeMode, Hour, Minute] in product A, B, and C. The query is:

**choose** [Timer, TapeMode, Hour, Minute] in {A, B, C}

The query of Example 2 returns the product B because the abstract state of B's operation-sequence is equivalent to the abstract state of the query (i.e. A: {time-setting}, B: {mode-setting, time-setting}, C: {time-setting, mode-setting}, and the query: {mode-setting, time-setting}).

# 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 describe the consistency management of *Visual CASE* system.

## 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 "GainMomentum" [Miy93][Syb94].

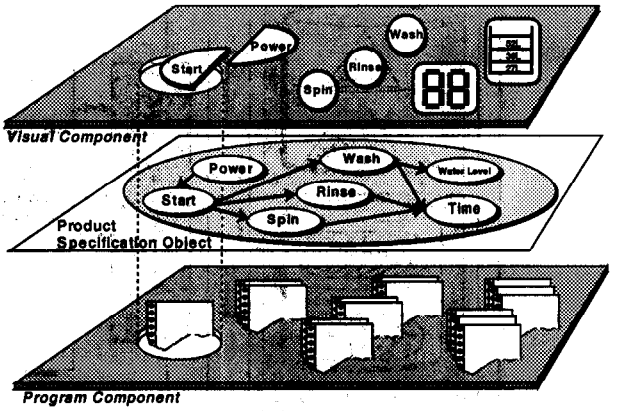


Figure 7: Visual Components and Program Components

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 7 shows the relationship of components and these subcomponents.

Figure 8 shows the architecture of *Visual CASE*. *Visual CASE* consists of six tools and five managers[4]. 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.

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

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

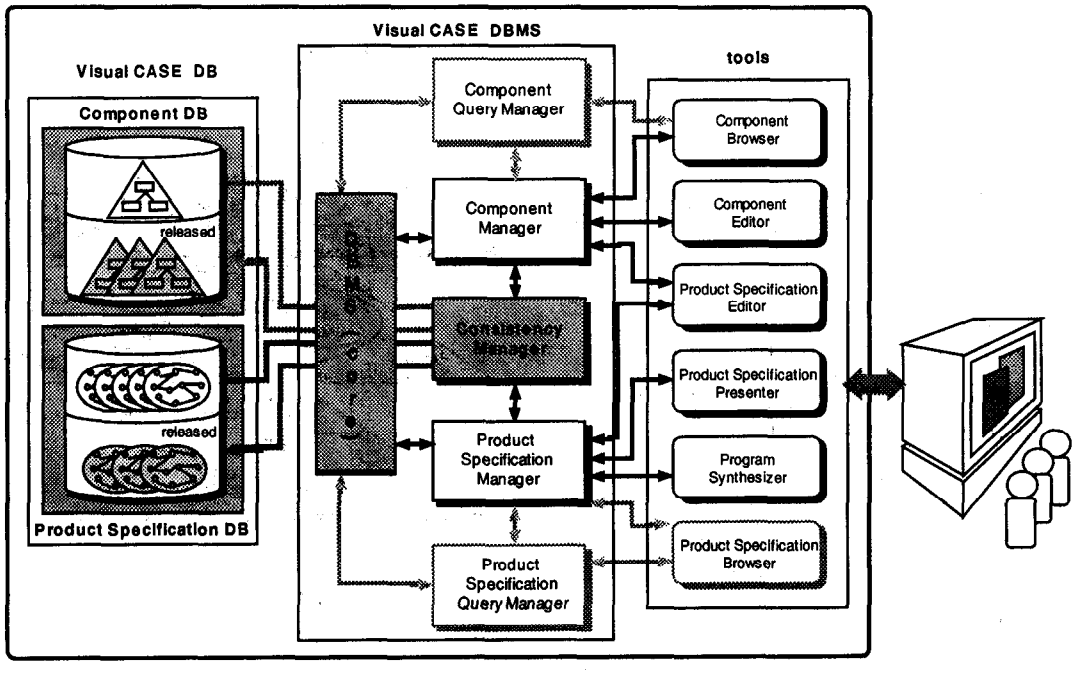


Figure 8: Architecture of Visual CASE

'pseudo' product on the CRT display. The **program synthesizer** generates a control skeleton of the target software. 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 **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 query manager** receive the request to search the component object and the product specification object from the component browser and product specification browser.

In Figure 1, the top part 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 editor for a particular component object to let the designers edit the component class definition. The view is divided into five sub-windows that display elements of the corresponding component object: the visual component, the program component, the specification parameters, the state attributes, the attributes, and the behavior. The bottom left part shows the view of the component browser to let the designers modify the component hierarchy.

### 4.2 Consistency Manager

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 consistency 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 consistency 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(core) is implemented on GainMomentum. As GainMomentum adopts Objectivity/DB[Obj90] as a storage manager, the DBMS(core) indirectly accesses Objectivity/DB through GainMomentum 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.

## 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, using program synthesizer.
3. The system can easily modify the specification with an interface through visual tools, using the *Visual CASE DB* as a product specification database.
4. The system can manage the evolution of the specifications by the release method while maintaining consistency.

There are several issues to be pursued about product specification database: long term transaction, view construction and so on. Our future work focuses on applying *Visual CASE DB* as a product specification database to actual software development of home appliances and verify the reusability and flexibility of our 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 Takeshi Nawata, Takuya Sekiguchi, Toshihiro Hishida, and Satoshi Kawabata. We would also like to thank Tim Cornish for polishing up the English, Chika Takayama and Yuko Hamada for drawing the figures in this paper.

## References

[GB95] V. Scott Gordon and James M. Bieman. Rapid prototyping: Lessons learned. *IEEE Software*, 12(1):85–95, January 1995.

[ISYH93] Y. Imai, K. Sumiya, K. Yasutake, and S. Haruna. Visual CASE: A Software Development System for Home Appliances. In *proceedings of the IEEE 17th International Computer Software and Applications Conference(COMPSAC93), Phoenix, AZ, U.S.A*, pages 11–18, November 1993.

[ITH92] K. Itoh, Y. Tamura, and S. Honiden. TransObj: Software prototyping environment for real-time transaction-based software system applications. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):5–30, March 1992.

[KA93] Halskov Kim and H. Peter Aiken. Experiences Using Cooperative Interactive Storyboard Prototyping. *Communications of the ACM*, 36(4):57–64, 1993.

[Kat90] R. H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.

[KBCG89] W. Kim, J. Banerjee, H. T. Chou, and J. F. Garza. Composite Object Revisited. In *proceedings of the ACM SIGMOD International Conference*, pages 337–347, June 1989.

[Mas91] Y. Masunaga. Design issues of OMEGA: An object-oriented multimedia database

management system. *Transaction of Information Processing Society of Japan*, 14(1):60–74, January 1991.

[Miy93] Y. Miyabe. Object-Oriented Multi-Media Application Development Software. In *proceedings of 8th German-Japanese Forum on Information Technology*, May 1993.

[NUF+95] Y. Nukina, W. Uchiyama, H. Fujii, Y. Omura, K. Iwamoto, and H. Tanaka. Washing machine with double cascades. In *National Technical Report*, pages 3–9. Matsushita Electric Industrial Co., February 1995. (in Japanese).

[Obj90] Objectivity. *Objectivity Database System Overview.* Objectivity Inc., 1990.

[Osb89] Sylvia L. Osborn. The Role of Polymorphism in Schema Evolution in an Object-Oriented Database. *IEEE trans. of Knowledge and Data Engineerings*, 1(3):310–317, September 1989.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[Sci91] E. Sciore. Multidimensional Versioning for Object-Oriented Databases. In *proceedings of Second International Conference on Deductive and Object-Oriented Databases*, December 1991.

[Shu91] Nan C. Shu. *Visual Programming.* Van Nostrand Reinhold, 1991.

[SOHI93] K. Sumiya, T. Ohtsu, S. Haruna, and Y. Imai. Visual CASE: An Object-Oriented Software Development System for Home Appliances. In *proceedings of 11th International Conference on Technology of Object-Oriented Languages and Systems(TOOLS USA'93)*, pages 97–107. Interactive Software Engineering, August 1993.

[Syb94] Sybase. *Gaim Momentum User's Guide.* Sybase, Inc, 1994.

[SYS+92] K. Sumiya, K. Yasutake, N. Sanada, S. Haruna, and Y. Imai. Query by operation-sequence: Extending query for product specification databases. In *Advanced Database System Symposium*, pages 53–61. IPSJ, December 1992. (in Japanese).

[TAU+94] H. Tanaka, S. Abe, W. Uchiyama, E. Ishizaki, T. Nawata, and Y. Imai. Prototyping System for Home Appliances – Case Studies in Control Panel Design. In *Proceedings of 14th Quality Management Symposium on Software Production*, pages 9–16. JUSE(Union of Japanese Scientists and Engineers), September 1994. (in Japanese).

[TNY+93] K. Tanaka, S. Nishio, M. Yoshikawa, S. Shimojo, J. Morishita, and T. Jozen. Obase object database model: Towards a more flexible object-oriented database system. In *Proceedings of the International Symposium on Next Generation Database Systems and Their Applications*, pages 159–166, September 1993.

[Zic91] R. Zicari. A Framework for Schema Updates In An Object-Oriented Database System. *IEEE ICDE91*, pages 2–13, 1991.