Wrapping Interactive Unix-Based Applications into CORBA Components

Jih-Woei Huang, Ping-Hung Kuo, Chih-Ping Chu[+]
Department of Computer Science and Information Engineering
National Cheng Kung University, Tainan, Taiwan 701, R.O.C.
E-mail: chucp@csie.ncku.edu.tw

Weng-Long Chang
Department of Management Information
Southern Taiwan University of Technology, Tainan, Taiwan 701, R.O.C.

Abstract

Component software is now the mainstream of software development due to its superior features in maintainability, reusability and productivity. Besides, distributed applications are currently very popular for the rapid development of the Internet and CORBA is one of the main underlying infrastructures to support distributed applications. Also, nowadays there exist a large number of legacy systems in the enterprises. Wrapping the legacy systems into reusable components is an economic way for the enterprises to enhance the information processing capability. In this paper, we present an interactive wrapper technique to make Unix-based interactive legacy systems act as CORBA components. We take Unix shell as the legacy system to realize the presented technique. Besides, we compose the wrapped Unix shell component with other self-developed CORBA components to build a 3-tiers distributed component software.

Keywords: Component Technology, Distributed Computing, Software Architecture, Software Integration, Software Reuse.

1. Introduction

Software reuse is an economic means to speed up the software development. As the way of manufacturing computer hardware, building software rapidly by means of assembling reusable components has been a goal for the software industry to strive. Generally, software component is an individual software piece with specific functions and can be viewed as a software IC, similar to its hardware counterpart. The software developed in the manner of assembling collaborative components, which interact with each other through consistent communication interfaces, is called component software (or componentware). The functions of the componentware are flexible to be changed by altering its components such that the componentware satisfies the need of software customization [6]. Due to superior features in maintainability, reusability and productivity, componentware is now the mainstream of software development.

Besides, the distributed applications are currently very popular due to the rapid development of the Internet. CORBA [13, 14] is, at present, one of the main middlewares (i.e., CORBA, DCOM and Java) to support distributed applications. It provides

the facilities to support the interoperations between components run on heterogeneous distributed environments.

Nowadays, there exist, especially in the enterprises, a large number of earlier developed application systems — legacy systems (legacy assets or legacy applications) that are designed to process enterprise information and are executed on their own specific operating environments. These legacy applications still play important roles in the enterprises and cannot be easily substituted. The information processing capability inside an enterprise, however, needs to be enhanced for satisfying the new business requirements. One of the ways to achieve is to redevelop the information systems from scratch with the emerging computer technologies, such as distributed-object technology, etc. Nevertheless, this will cost plenty of development time and money. As mentioned, software reuse is an effective means to speed up software development. Reusing the legacy systems in developing new application systems can apparently save software development cost. Furthermore, enabling the legacy systems to act as components can make them integrated or composed with other components on heterogeneous environments through the support of middleware so as to come up to the new trend of software development.

There have been works [2, 4, 7] to present the methodologies or procedures for enabling the legacy applications to turn out to be the components. However, it is not an easy work to make legacy systems into components due to that most of them are in binary codes form — we have no way to access their source codes. Generally, the wrapper technique for software integration can be used for this. The primary notion behind the wrapper is to present the functions of the legacy systems according to the component communication protocol supported by the middleware.

There are many legacy systems that are Unix-based and operationally interactive. But how to wrap the Unix-based interactive legacy systems into components are scarcely discussed. In this paper, we present the interactive wrapper technique to make interactive Unix-base legacy systems into CORBA components. We take Unix shell as the sample legacy system to practice the proposed technique. In addition, the Unix shell component is composed with other self-developed CORBA components to build a 3-tiers distributed componentware.

The rest of this paper is organized as follows. In Section 2, the related works is introduced. In Section 3, our interactive wrapper technique is detailed discussed. In Section 4, the implementation of the 3-tiers distributed componentware is presented. Finally, a brief conclusion is drawn in Section 5.

2. Related Works

The wrapper is the key technique for software integration of binary code level. Most of the proposed wrappers relate to data integration or need the support of well-defined programmatic interfaces [8, 11, 12]. Whereas, the wrapper in the FIM (Function Integration Model) [9] intercepts and redirects the command/data streams between the integrated applications such that the whole or part functions of these applications are cooperatively performed. The FIM wrapper can integrate both the data and the functions of different applications without specific tool-supported APIs.

On the other hand, the CFIM (CORBA Function Integration Model) [1, 10] extends the FIM to enable an application to act as a CORBA object. The wrapper (CORBA wrapper) in the CFIM serves as an interface that presents the non-CORBA based applications as the CORBA services. In addition to being responsible for redirecting messages to the wrapped application, the wrapper needs to provide CORBA interfaces for it. The architecture for CFIM-based system integration is shown in Figure 1 [10].

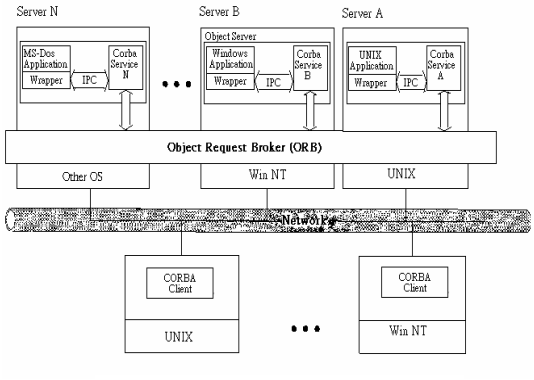

Figure 1. The CFIM diagram

To serve the requests from CORBA client, the wrapper needs to fulfill the following operations:

1. Receive the CORBA requests and transform them into corresponding command/data streams accepted by the wrapped application.
2. Transfer the command/data streams to the wrapped application.
3. Receive the result from the wrapped application and transform into corresponding CORBA messages.
4. Return the CORBA messages back to the client.

On the whole, as shown in Figure 2, the CORBA wrapper includes two main parts: an I/O redirector to redirect the command/data streams to and out of the wrapped application; a CORBA interface adaptor to present the outsides the CORBA interfaces of the wrapped application, the internal implementation of which is to receive, transform and convey the CORBA messages or the messages of the wrapped application.

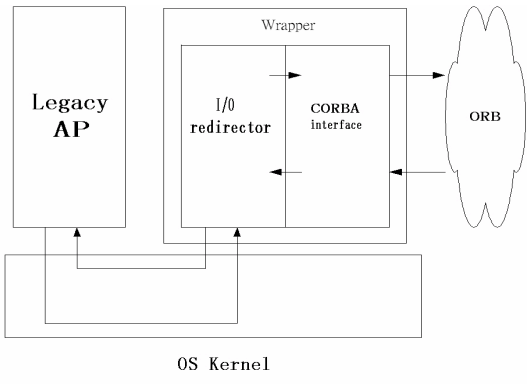

Figure 2. The architecture of the wrapper

Based on this CORBA wrapper, pipe is employed to be the communication infrastructure between the I/O redirector and the wrapped application [1]. However, due to the operational limitation, the pipe is not suitable to be that between the I/O redirector and the wrapped application which originally communicates with the users interactively. We will discuss this in next section.

3. The Interactive Wrapper

In this section, we present the interactive wrapper for making Unix-based interactive legacy systems into CORBA components.

3.1 The I/O redirector

It is obvious that a communication link needs to be established to redirect the command and data streams between the I/O redirector and a legacy

system. In Unix, this can be achieved by employing a specific IPC (Inter-Processes Communication) mechanism. In terms of the Unix, the I/O redirector and the legacy system are coprocesses that communicate through the IPC, and the output of the former is the input of the latter and vice versa, as shown in Figure 3.

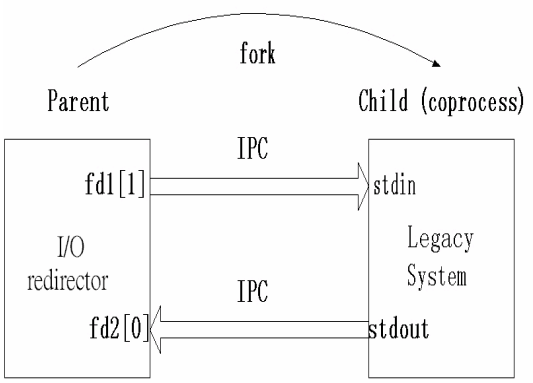

Figure 3. The coprocesses diagram

Unix I/O Buffering

To lessen the number of reading/writing the input and output data stream so as to promote the system performance, the Unix standard I/O library offers various I/O buffering mechanisms: fully buffered, line buffered, and unbuffered [15, 16]. Essentially, the fully buffered will be adopted if the input and output data streams are operationally related to the interactive device; otherwise, the line buffered will be used. The unbuffered is mainly employed in connection with the standard error.

Pipe

There are many diverse IPC mechanisms, e.g., pipe, FIFO, message queue, shared memory and socket, to be provided by different operating systems [15, 16]. In Unix, pipe that connects the standard output channel of one process (the writer process) to the standard input channel of another process (the reader process) is a commonly used IPC mechanism. Figure 4 shows the typical Unix pipes between two processes.

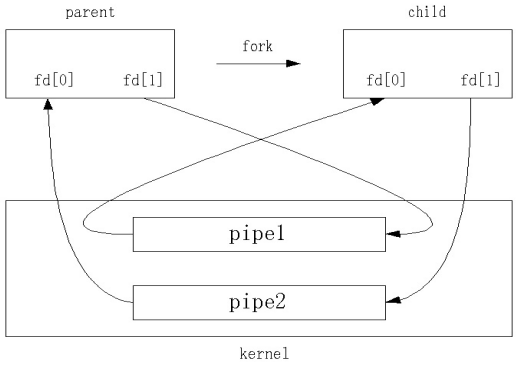

Figure 4. The Pipes between coprocesses

Being an IPC mechanism between two processes, the pipe works well if the two processes communicate with each other through it for once. However, utilizing the pipes as the IPC mechanism between the I/O redirector and the interactive legacy system will cause troubles.

Firstly, an interactive legacy system originally interacts with the user by means of the terminal. The user inputs the command/data from the keyboard, the legacy system eventually receives and processes the request and returns the result back to display on the monitor. If the I/O redirector transmits command and data to the interactive legacy system through a pipe and returns the result to the user through another pipe, the “stdin” and “stdout” of the legacy system is apparently connect to the pipes rather than the real terminal.

Besides, if the pipe is adopted as the IPC mechanism and the legacy system employs the standard I/O functions to read/write the “stdin “and “stdout”, then the standard input and standard output

will be operated in the manner of fully buffered. The fully buffered type requires the writer process (I/O redirector) to be terminated for enabling its output to be read by the reader process (legacy system). This will disable the I/O redirector to continuously interact with the legacy system. Thus, the pipe is not suitable to be the IPC mechanism between the I/O redirector and the interactive legacy system since they generally need to be continuously kept alive and interactively transfer data to each other.

Pseudo terminal

Instead, the pseudo terminal [17], which adopts the line buffered as the buffering type that requires no termination of the writer process and thus allows the reader process to interactively interact with the writer process, is suitable to be the IPC mechanism between the I/O redirector and the interactive legacy system. Due to the interactive communications between the legacy system and the outsides, the pseudo terminal that acts as a terminal to an application but is not a real terminal is adopted to be the IPC infrastructure in our implementation of the I/O redirector, as shown in Figure 5.

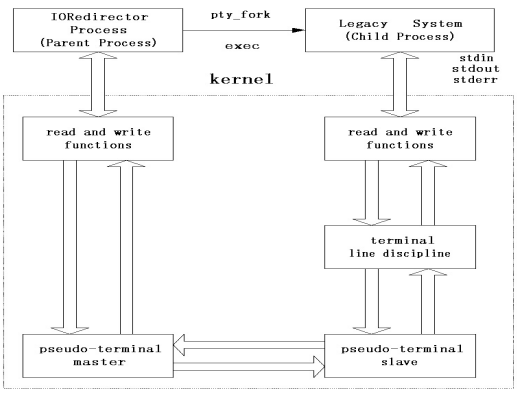

Figure 5. The pseudo terminal diagram

The followings are the main operational features of the pseudo terminal:

1. A parent process (I/O redirector) first invokes “open” system call to open a pseudo-terminal master process and then calls “fork” to generate a child process. The child process sets up a new session, invokes “open” system call to open the corresponding pseudo-terminal slave process, duplicates the slave to be its standard input, standard output and standard error device, and finally calls “exec” to execute the interactive legacy system. Thus, the pseudo-terminal slave becomes the controlling terminal of the legacy system.
2. In this way, the pseudo-terminal slave appears to be a terminal device related to the standard input, standard output and standard error of the legacy system. It can issue all the terminal I/O functions on these descriptors. But since it is not an actual terminal device, some functions that make sense to real terminal (e.g., change the baud rate, send a break character, etc.) will be ignored.
3. Anything written to the master pseudo terminal is considered to be the input to the slave pseudo terminal and anything out of the slave appears to be the output of the master. The pseudo terminal looks like a stream pipe, but with the terminal line discipline module above the slave it has additional capabilities over a plain pipe.

3.2 CORBA interface adaptor

Generally, the major effort to provide CORBA interfaces for the legacy system with the source codes available is to define its functions by means of the corresponding CORBA IDL definitions. However, we are now devoted to provide CORBA interfaces for the legacy system with only binary codes

available. The major work for this mainly involves: converting the CORBA messages into legacy system message and vice versus; conveying messages to the I/O redirector for being redirected to the legacy system or to the ORB for being returned back to the CORBA client. In our wrapper technique, an interface adaptor is designed to achieve this. In our design, the interface adaptor presents a single CORBA IDL definition to the CORBA ORB. In this way, the interface adaptor encapsulates the legacy system as CORBA component. The CORBA IDL definition the interface adaptor presents is as follows:

```
module Wrap
{
   interface IO
  {
     unsigned short start(in string APname);
     boolean redirector(inout string, APstream, in
     unsigned short APpid, in unsigned short flag);
       };

};
```

There is only one CORBA interface definition — "*IO*" in the interface adaptor. This interface definition includes two methods — *start* and *redirector*.

The start method definition includes only one parameter, *APname*, of which the type is CORBA::string. The keyword "*in*" means that the server (i.e., ORB) write data onto APname for the client (i.e., interface adaptor) to read. The start method is for invoking the legacy program with path name specified in APname. The start method will return a value with type of CORBA::unsigned short to represent the process ID of the invoked legacy program.

The redirector method definition contains three parameters: *APstresm* is used for transmitting data stream to and out of the legacy systems. The keyword "*inout*" means that the server can write data onto APstream for the client to read and vice versus; *APpid* is used to store the process ID of the invoked legacy program; *flag* is used to store the status of the legacy systems (1 means invoked just now; 0 means already in execution). The redirector method will convert the format of the messages before conveying them to the legacy system or the ORB.

Due to the interactive feature of the legacy system we desire to wrap, except the external IPC mechanism between the I/O redirector and the legacy system, the internal IPC mechanism between the interface adaptor and the I/O redirector needs to support interactivity. Unlike the legacy system in binary codes form, we take hold of the inside of the wrapper. To satisfy the requirement, we adopt FIFO (First In First Out) to be the interactive IPC mechanism between the interface adaptor and the I/O redirector in our wrapper. We create two FIFOs, of which the file names are the process ID of the legacy system, to manage the bi-direction flow of the command/data streams. The architecture of the internal and external IPC mechanisms of our wrapper is shown in Figure 6.

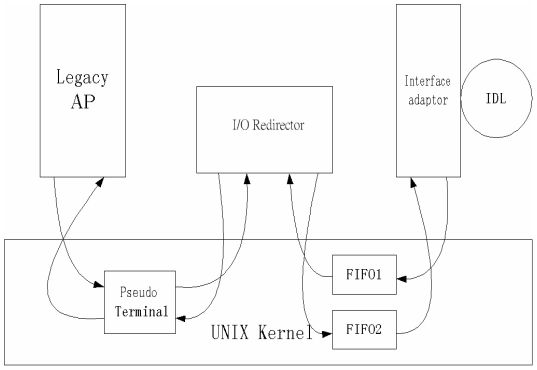

Figure 6. The IPC mechanisms of the proposed wrapper

4. A 3-tiers Distributed Componentware

We have taken the Unix shell as the legacy system to realize the presented technique. Besides, we build a CORBA-based distributed application to experience the CORBA capability of supporting distributed heterogeneous environments. This application can provide the user the information of current date, current week and current month. The user can query these information through selecting the service options on the user interface which is execute on the MS-DOS environment.

The 3-tiers software architecture currently is the mainstream of the architecture of the distributed application [5]. It extends the traditional 2-tires (client/server) software architecture by inserting an additional application layer (or business logic layer) between the traditional user interface and database layers.

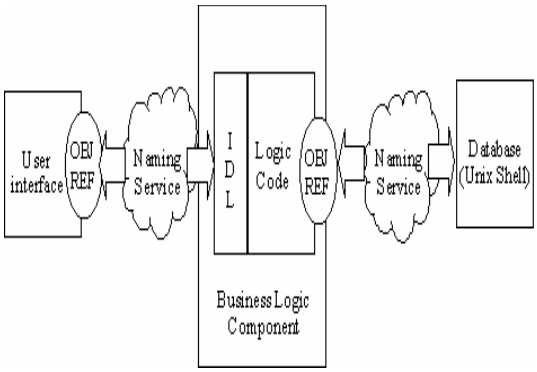

Figure 7. The 3-tires distributed componentware

From the viewpoint of the componentware, the user interface components, the business logic components and the database components can be assembled into larger versatile software applications. As shown in Figure 7, the developed componentware is a 3-tiers distributed application.

User Interface Component

We experimentally construct a simple user interface component to provide the user three main options — “date”, “week” and “cal” to query, respectively, current date, current week and current month. Besides, an “exit” option allows the user to quit the application. This user interface component is executed on the MS-DOS environment.

Business Logic Component

As mentioned, this application provides the user three services — to query the information of current date, current week and current month. The first and the last services can be directly achieved by utilizing the existing Unix utilities — *date* and *cal*. However, the information of current week cannot be directly obtained but can be produced by combining the services of the previous two utilities. Thus, we construct an additional CORBA component to be the business logic component which can individually request the *date* or *cal* service from the wrapped Unix shell as well as integrate both services. If the user request is to query current date or current month, this component will transmit *date* or *cal* request to the underlying wrapped Unix shell component (or database component, as describe below) for acquiring the information of the current date or current month. If the user request is to query current week, this component will transmit *cal* and *date* request in turn to the underlying database component, calculate the current week from the returned data and return the current week to the user interface component. The CORBA IDL of this business logic component can be referred to in the appendix.

Database Component

In terms of the 3-tiers architecture application, the wrapped Unix shell component is taken for the database component in this application. Originally, the Unix shell accepts the input command and data from the users and then return the result data as output to the users. In this way, the Unix shell behaves as a database to some extent and the input command and data are viewed to be the query to the database. When the wrapped shell component receive *date* or *cal* request, it will execute *date* or *cal* utility to acquire the information of current date or current month and return the information back to the business logic component.

Figure 8 shows the functional architecture of this distributed application.

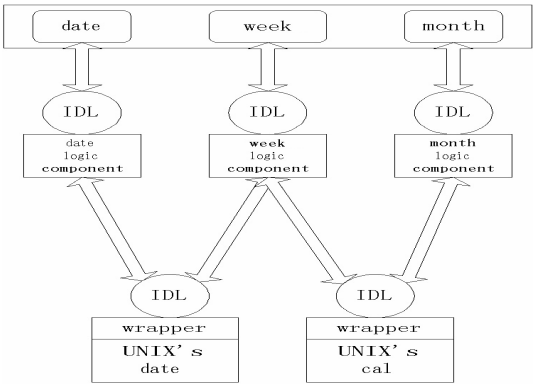

Figure 8. The functional architecture of the distributed calendar application

Software IC description file

As described before, the functions of the componentware are flexible to be changed. From the viewpoint of the user interface component, the services it presents to the user are offered by the business logic components (or the wrapped legacy components eventually). Properly information related to these services-providing components can be specified to describe the presented services, and altering the information with relation to these components means to change the functions of the componentware. On the other hand, each software component can be viewed as a software IC as described before. Thus, we define a set of pin-like descriptions for each of the software IC in a software IC description file for this purpose. While the application begins to be executed, the user interface component will first read the software IC description file and then initialize the corresponding components for future collaboration. The software IC description and our description file for this case can be referred to in the appendix. The test result of this application can be referred to in [3].

5. Conclusion

Enabling the legacy systems to act as the reusable components can facilitate the development of new applications in an economic way and comes up to the new trend of software development as well. In this paper, based on the CORBA Functional Integration Model, we present an interactive wrapper technique to make Unix-based interactive legacy systems into CORBA environments. The key technique in this model is the CORBA wrapper that includes two main parts — an I/O redirector to redirect messages between the wrapped application and the outsides; a CORBA interface adaptor to provide CORBA interfaces for the wrapped application. Previous researches on software integration utilize pipe as the underlying IPC mechanism between the I/O redirector and the wrapped application. Due to the operational limitation, pipe is not

suitable to be the IPC mechanism for two interactive processes to communicate on. Instead, pseudo terminal, which is not a real terminal by which the user interactively interacts with the application in real world but acts as a terminal to an application, is employed to be that in our wrapper to support the interactivity.

From our componentware research, we learn that the application developer can produce, according to the user requirement, new business logic components by appropriately combining the services provided by the wrapped legacy components which can be considered to be database components in terms of the 3-tiers software architecture. In addition, we find that by properly assembling the CORBA user interface components, business logic components and database components, the componentware can be built to be distributed application of 3-tiers architecture through the support of CORBA middleware.

Reference

1. 吳大欣, 林志敏, 焦惠津, "利用包裝程式技術達成 CORBA 環境下 UNIX 應用程式之重用", 第八屆物件導向技術及應用研討會.
2. 柯仁傑, 洪麗玲, 陳茂華, "以元件組裝方式開發"舊系統上網工具"之經驗報告", 資訊工業策進會軟體工程實驗室,第八屆物件導向技術及應用研討會.
3. 郭炳宏, "UNIX 作業環境應用軟體再利用之研究", 國立成功大學碩士論文, 2001.
4. Paul Allen, Stuart Frost, “Component-Based Development for Enterprise Systems”, Cambridge, ISBN 0-521-64999-4.
5. Israel Ben-Shaul, Gail Kaiser, "Coordinating Distributed Components Over The Internet", IEEE Internet Computing, Volume: 2 Issue: 2, pp. 83-86, March-April 1998.
6. Klaus Bergner, Andreas Rausch, Marc Sihling, “Componentware-The Big Picture”, http://www.sei.cmu.edu/cbs/icse98/papers/p6.html.
7. Ted J. Biggerstaff, "Design Recovery for Maintenance and Reuse", Computer, Volume: 22 Issue: 7, pp.36-49, July 1989.
8. Y. S. Lee, Rainbow: Prototyping the DIOM Interoperable System, http://www.cse.ogi.edu/DISC/DIOM/yoosh/goodbye/html/goodbye.html.
9. Jim-Min Lin, “Cross-Platform Software Reuse by Functional Integration Approach,” Proceedings of COMPSAC 97, pp.402-408, Aug 1997.
10. Re-Chi Lin, Jim-Min Lin, Hewi Jin Jiau, "Reusing MS-Windows Software Applications Under CORBA Environment,” Proceedings of 1998 International Conference on Parallel and Distributed Systems, pp.615-622.
11. M. T. Roth and P. Schwarz, A Wrapper Architecture for Legacy Data Sources, http://www.almaden.ibm.com/cs/garlic/vldb97wraprj.ps.
12. A. Sahuguet and F. Azavant, WysiWyg Web Wrapper Factory, http://ww.cis.upenn.edu/~sahuguet/WAPI/wapi.
13. Douglas C. Schmidt, Steve Vinoski, “An Overview of the OMG CORBA Messaging Quality of Service Framework”, OMG. Http://www.omg.com/.
14. Douglas C. Schmidt, Nanbor Wang, Steve Vinoski, “Collocation Optimizations for CORBA”, OMG. Http://www.omg.com/.
15. W. Richard Stevens, ”UNIX NETWORK PROGRAMMING Volume 1-Networking APIs:

Sockets and XTI 2nd", Prentice Hall, ISBN 0-13-649328-9.

16. W. Richard Stevens, "UNIX NETWORK PROGRAMMING Volume 2-Interprocess Communications 2nd", Prentice Hall, ISBN 0-13-020639-3.
17. W. Richard Stevens, "Advanced Programming in the UNIX Environment", Addison-Wesley Professional Computing Series, ISBN 0-201-56317-7.

Appendix

1. The IDL definition of the business logic component:

```
module Logic
{
  interface basic_interface
  {
    unsigned short start(inout string APname, in string
                         CompRelation);
    boolean redirector(inout string APstream, in
    unsigned short APpid, in unsigned short flag);
  };
};
```

Parameter *CompRelation* in the start method is used to note the order to combine components such that the logic component can be aware of the next component to be combined.

2. The software IC description:

a. #SOFT_IC_DEF: the start of the software IC description.
b. SOFT_IC_NAME: the software IC (function) name related to the service on the user interface.
c. SOFT_IC_TYPE: the operating platform on which the software IC is executed.
d. SOFT_IC_PATHNAME: the execution path of the software IC.
e. SOFT_IC_ENTRY: the entry point of a certain function of the software IC.
f. SOFT_IC_LOGIC: the (logic) components to provide the services.

3. The software IC description file for this case:

#SOFT_IC_DEF // Week service description
SOFT_IC_NAME：week
SOFT_IC_TYPE：Linux
SOFT_IC_PATHNAME：cal，date
SOFT_IC_ENTRY：
SOFT_IC_LOGIC：Week

#SOFT_IC_DEF //Date service description
SOFT_IC_NAME：date
SOFT_IC_TYPE：Linux
SOFT_IC_PATHNAME：date
SOFT_IC_ENTRY：
SOFT_IC_LOGIC：IORedirector

#SOFT_IC_DEF //Month service description
SOFT_IC_NAME：cal
SOFT_IC_TYPE：Linux
SOFT_IC_PATHNAME：cal
SOFT_IC_ENTRY：
SOFT_IC_LOGIC：IORedirector