Re-engineering Software Architecture of Home Service Robots: A Case Study

Moonzoo Kim, Jaejoon Lee,
Kyo Chul Kang
Computer Science and Engineering Department
Pohang University of Science and Technology
Pohang, South Korea
{moonzoo,gibman,kck}@postech.ac.kr

Youngjin Hong, Seokwon Bang
Interaction Lab.
Samsung Advanced Institute of Technology
P.O.Box 111, Suwon
440-600, South Korea
{bhong,banggar.bang}@samsung.com

ABSTRACT

With the advances of robotics, computer science, and other related areas, home service robots attract much attention from both academia and industry. Home service robots present interesting technical challenges to the community in that they have a wide range of potential applications, such as home security, patient caring, cleaning, etc., and that the services provided by the robots in each application area are being defined as markets are formed and, therefore, they change constantly.

Without architectural considerations to address these challenges, robot manufacturers often focus on developing technical components (e.g., vision recognizer, speech processor, and actuator) and then attempt to develop service robots by integrating these components. When prototypes are developed for a new application, or when services are added, modified, or removed from existing robots, unexpected, undesirable, and often dangerous side-effects, which are known as feature interaction problem, happen frequently. Reengineering of such robots can make a serious impact in delivery time and development cost.

In this paper, we present our experience of re-engineering a prototype of a home service robot developed by Samsung Advanced Institute of Technology. First, we designed a modular and hierarchical software architecture that makes interaction among the components visible. With the visibility of interactions, we could assign functional responsibilities to each component clearly. Then, we re-engineered existing codes to conform to the new architecture using a reactive language Esterel. As a result, we could detect and solve feature interaction problems and alleviate the difficulty of adding or updating components.

Categories and Subject Descriptors

D.2.11 [**Software Architecture**]

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.
Copyright 2005 ACM 1-58113-963-2/05/0005 ...$5.00.

General Terms

Design, Reliability

Keywords

software re-engineering, robot programming, reactive systems

1. INTRODUCTION

Home service robots have received much attention from academia as well as industry in anticipation that home service robots will potentially increase quality of human life in a wide range of application areas. Thus, leading consumer product companies such as Sony [2], Honda [1], and Samsung have invested a great deal of efforts in developing home service robots. Home service robots utilize various technology-intensive components such as speech recognizers, vision processors, and actuators to offer service features. As markets for home service robots are still being formed, however, these technical components undergo frequent changes and new services are added or existing services are often removed or updated to address changing needs of the user. Thus, home service robots present interesting software engineering challenges to the research and development community.

Due to limited development resources, developers of home service robots tend to focus on technology intensive components at an early stage of product development without an architectural consideration of how they will be integrated to create services. Furthermore, engineers are often grouped into separate teams based on the underlying technologies (e.g., speech processing, vision processing), which makes integration of these components more difficult. Without a fore-thought architectural design, an initial product tends to be developed by integrating these components in an adhoc and bottom-up way. As a consequence, products often suffer from feature interaction problems [10, 24]. Feature interaction problems found in systems developed without careful architectural design are hard to analyze and solve because it is difficult to see how behaviors of components are coordinated to create services, i.e., how control information flows between components for services. Therefore, in order to improve quality of the product, *re-engineering* of the product had to be enforced at a later stage.

In this paper, we describe our experience of re-engineering a prototype of a home service robot, called Samsung Home Robot (SHR), developed by Samsung Advanced Institute of

Technology (SAIT). We concentrated on designing a software architecture (SA) [15, 23, 5] for easy integration of components. More specifically, we focused on a SA that makes interactions between components visible, thus allowing behavior analyses of a product. We designed a SA for SHR based on the following three principles.

- separation of the control plane from the computational plane
- distinction between global behavior and local behavior
- layering in accordance with a data refinement hierarchy

The new SA designed with these principles is *modular* and *hierarchical*, and systematic addition or modification of service components became feasible. With the interaction visibility between components, we could assign behavioral responsibilities to the components accurately, and detect/solve feature interaction problems easily. We re-engineered the existing implementation to conform to the new SA using a reactive language Esterel. Because of the compositionality property and reactive operators of Esterel, the re-engineered implementation became compact and easy to analyze and update.

Section 2 describes components and services of SHR and reviews the previous implementation of SHR. Section 3 introduces the three principles used for re-engineering, and the new SA created by applying these principles and experiment results are described in section 4. Section 5 summarizes the lessons learned from the project, and section 6 concludes this paper with future works.

2. BACKGROUND ON SHR

SHR is a prototype of a home service robot for daily home services such as home surveillance, etc. Section 2.1 explains history of developing SHR. The hardware of SHR is described in Section 2.2. The services of SHR are explained in Section 2.3. Section 2.4 shows the previous architecture of SHR. Finally, Section 2.5 describes the statistics on the SHR software.

2.1 Development History

SHR100 is a successor of SHR50 and SHR00. Development of SHR00 started in 2002 by four separate teams consisting of 13 people working on speech recognition, vision recognition, map building, and actuator control. These teams completed developing their own part and tried to integrate their parts altogether at a later stage. SHR50 as well as SHR00, however, exhibited often unstable behaviors such as missing user commands and stuttered movement although each part had worked successfully when not integrated (this kind of failure is not uncommon in robotics field [13]). As a consequence, they decided to give up SHR50 and SHR00 and develop both hardware and software of SHR100 from scratch. To prevent similar problems, SHR100 was equipped with larger memory and faster CPU. Also, SAIT requested POSTECH to design a software architecture after ten months of the new development (at that point, SAIT completed a high-level task specification of SHR100 and several service features were implemented). At that request, POSTECH reviewed the specifications as well as the implementation of SHR100. Then, POSTECH designed a new software architecture and re-engineered existing implementation for six months.

2.2 Hardware

SHR100 has a single board computer (mobile Pentium IV 2.4G with 512MB memory running embedded WindowsXP) controlling peripherals as follows.

- Input peripherals
  - 1 ceiling camera for building a map (640x480 resolution and 5 frames/s)
  - 1 front camera for recognizing users and remote surveillance (320x240 resolution and 15 frames/s)
  - 8 microphones for speaker localization and speech recognition (8 Khz sampling rate)
  - 1 structured light sensor for obstacle detection and footstep recognition
- Output peripherals
  - 1 LCD display for information display
  - 1 speaker for speech generation
  - 2 actuators for right and left wheels
- Input/output peripheral
  - Wireless LAN for communicating to a home server

The components of SHR100 are illustrated in Figure 1.

2.3 Services

Some of the primary services of SHR100 are described as follows.

- Call and Come (CC)
  This service first analyzes audio data sampled from eight microphones attached to the surface of the robot and detects predefined sound patterns (e.g., hand clap or voice command). There are two commands “come” and “stop”. Once a “come” command is recognized, the robot tries to detect the direction of sound source by comparing the strength of sound captured by eight microphones. Then, the robot rotates to the direction of sound source and tries to recognize a human face by analyzing video data captured by the front camera. If the caller’s face is detected, the robot moves forward until it reaches within 1 meter from the caller (distance from the caller is measured by the structured light sensor).

  A “Stop” command is similar to a “come” command except that the robot does not move forward if a “stop” command is given while the robot is not moving. When the robot is moving, a “stop” command makes the robot stop. If command recognition, sound source detection, or face recognition fails, CC resets to the initial state and waits for new commands. CC is preemptible, i.e., newly recognized command makes the robot ignore the previous command and follow the new one.

- User Following (UF)
  The robot uses the front camera and the structured light sensor to locate a user. Once UF is triggered, the robot constantly checks vision data and sensor data

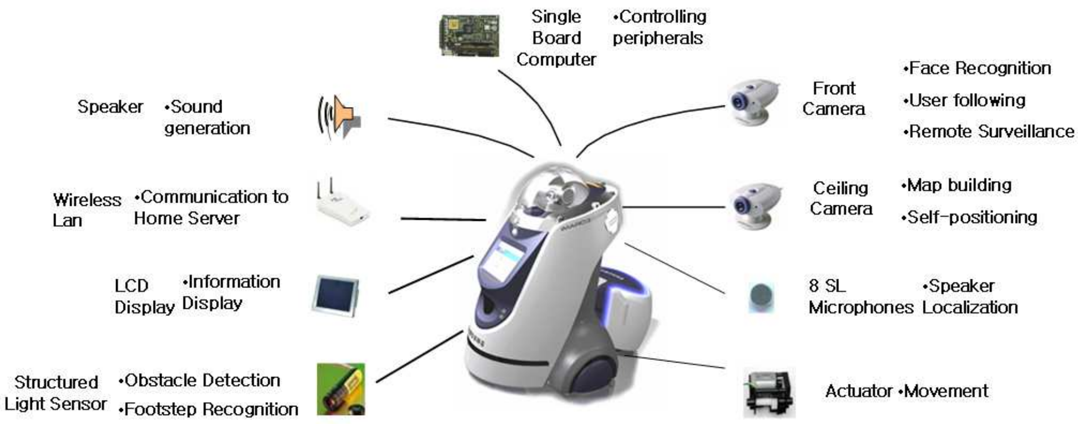


Figure 1: Hardware components of SHR100

from the structured light sensor in every 200 ms for locating the user. The robot keeps following the user within 1 meter range. If the robot misses the user, the robot notifies the user by speaking “I lost you” and UF terminates. Then, the user gives “come” command to let the robot to recognize the user and restart UF. Similar to CC, UF is a preemptible service.

- Security Monitoring (SM)
  The robot patrols around a house for surveillance using the map generated by Simultaneous Localization and Map building (SLAM) module. Intrusion or accidents are defined as patterns recognizable from vision and sound data. For example, intrusion can be detected by watching images and sounds from doors and windows. Once such an event is detected, the robot notifies the user directly via an alarm or indirectly through a home server.

- Tele-presence (TP)
  A remote user can control the robot using a PDA. The robot sends the remote user a map of the house generated by the SLAM module periodically. The user can command the robot to move to a specific position in the map displayed at the PDA. In addition, the robot can send images obtained from the front camera to the remote PDA for surveillance.

2.4 Architecture

Figure 2 illustrates the previous SA of SHR100. Each service component in Figure 2 implements service feature mentioned in Section 2.3. The input data are gathered from the sensors (e.g., the 8-channle microphones, the front camera, etc) and distributed to the corresponding service components. The service components read and process the input data to retrieve information required for the services. For instance, CC component processes the audio data to identify the user’s commands and also to detect the direction of the sound source. The outputs of the service components are action commands (e.g., rotate, go forward, stop, etc) moving the robot to specified directions. The navigation component receives these commands and converts them into schema data to control the actuators. While the robot is moving, it also performs the reactive actions such as obstacle avoidance or emergency stop, which are critical for safety.

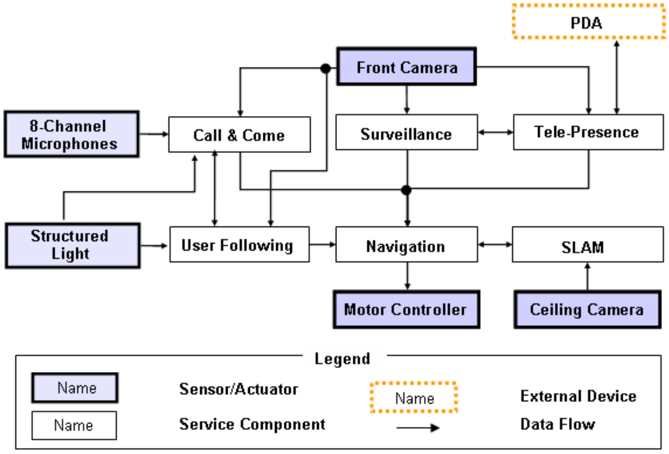


Figure 2: Previous SA of SHR100

When only a few service components are integrated, their interactions are trivial and manageable with this architecture model. As more services are integrated, however, the complexity of their interactions grows exponentially in this architecture. Thus, issues such as priorities among services or global system modes are hard to handle correctly.

2.5 Software Statistics

The version of the SHR100 software which POSTECH re-engineered contained complete CC and UF components. Other services such as security monitoring or tele-presence were not completely implemented in the version of the SHR100 software. The rough statistic summary of the SHR100 software is described in Table 1. For the CC and UF services, and other related parts, we show the total number of source files and total lines of code. Critical parts of the application (mostly recognition algorithms and device drivers) were given to POSTECH from SAIT as DLL (Dynamic Linked Library) format for security reason. For DLLs, we show the number of DLL files and total size of DLLs.

Components	# of source files	Size
Call and come	29	4000 lines
User following	43	9000 lines
Others	43	3600 lines
DLLs	39	38 MB

Table 1: Statistics on the SHR100 application

3. RE-ENGINEERING PRINCIPLES

First, we reviewed specification and implementation of SHR100 thoroughly. Based on observations from the reviewing process, we could propose three re-engineering principles. Figure 3 illustrates a new SA designed according to these three principles.

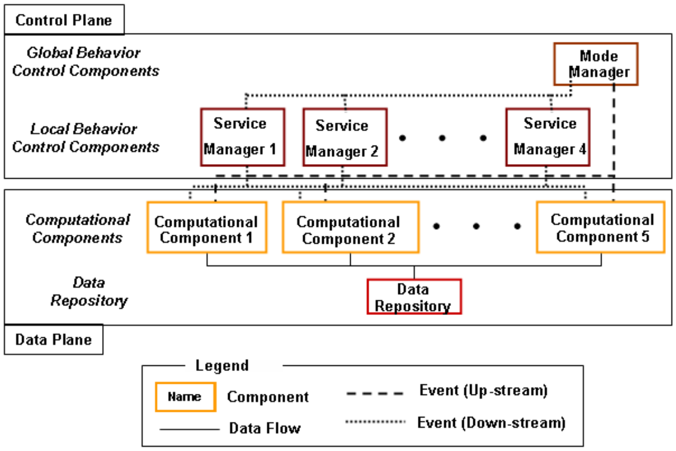


Figure 3: SA designed based on the principles

3.1 Principle 1: Separation of Control Components from Computational Components

There are two classes of data manipulated by SHR100. The first class of data is computational data (voice/vision/sensor data) which are handled in large volume. The second class of data is control data for controlling components. These two classes of data have distinct characteristics. For example, missing a few vision images may result in less accurate vision recognition, but probably not a critical failure. Losing a single stop control signal, however, may cause damage to the valuable properties of a house.

By clearly separating the control plane containing control components from the data plane containing computational components, data flow among components can be classified more clearly because we can distinguish control data flow from computational data flow. Furthermore, we can apply different development methodologies optimized for each plane. In other words, we can apply control oriented development methodology to the control plane and data oriented development methodology to the data plane, which increases reliability and efficiency of the system.

For the control plane, *correctness* is the foremost concern due to complexity of reactive systems. Therefore, for the control plane, adopting a formal method framework such as Esterel [6] to design, implement, and validate/verify is a suitable way [14]. [1] [22] studies four different formal methodologies for robotics domain. MAESTRO [12] and ORCCAD [7] provide high-level languages specialized for robotics domain. For the data plane, efficient computation is the most important goal. In addition, computational components need to communicate with hardware devices such as camera and microphone. Therefore, the data plane is implemented in C/C++ and assembly language for both efficiency and communication with HW.

3.2 Principle 2: Separation of Global Behaviors from Local Behaviors

When a home service robot is developed by integrating components implementing various features, these features may interact with each other. Without careful analysis of their interactions, however, they may cause feature interaction problems. The main cause of feature interaction problems is the unclear separation of *global* concern from *local* ones. Each service feature may have its own state (local concerns) to provide the service (e.g., UF may have 'user relocating' state). At the same time, the global concerns should also be defined and maintained for the system integrity (e.g., the safety critical user commands should override currently active services).

The integration of SHR100 had been made in an ad-hoc and bottom-up manner and the robot sometimes exhibited unexpected behaviors. For instance, we noticed that the robot occasionally ignored a "stop" command during the UF service: this was a safety critical problem and we tried to identify its cause. Through the analysis, we found that a feature interaction problem had occurred between the UF and CC features. Basically, UF was designed to track a user only with vision data, not with audio data. Therefore, when UF failed to locate a user the robot was following, UF requested CC to relocate the user by detecting the direction of the sound source. The feature interaction problem described above had occurred at this situation, when the user's voice command was "stop". The CC feature sent the direction of sound source to UF and also sent a stop signal to the Navigation component. However, UF resumed moving the robot to the direction of the user informed by CC.

To address such problems, we have applied the second engineering principle for designing the control plane. Each of the Service Manager components in Figure 3 defines the behavior of service feature by controlling the computational components. Also, each service component can be executed and tested independently from other service components. The Mode Manager component defines the system modes (e.g., initialization, termination, power saving, and charging modes) and the interaction policy (e.g., priority, concurrency) between services features.

Mode Manager monitors information from components and defines global states based on interactions among the components. Based on global state, Mode Manager sends controlling signals (e.g. suspend, resume, and reset, etc) to service components. Service components should have interfaces for these controlling signals. In other words, policies on services can be enforced using these controlling signals. With this architecture model, we could specify, modify, and validate the robot's behavior specifications systematically.

[1] Formal framework for reactive systems is hard to apply toward mixed architecture such as Figure 2 because formal framework focuses on control aspects.

3.3 Principle 3: Layering in Accordance with Data Refinement Hierarchy

We also analyzed the previous architecture with respect to data computations required for service features. We found that there existed computational redundancies among services. For example, the image format conversion component, which converted captured image data into a file format (e.g., JPEG or GIF), was required by the UF, CC, SM, and TP services. With the previous architecture, the image format conversion component had to be replicated at each of the service components, which resulted in high consumption of resources such as CPU, memory, etc.

Another finding was that data computations could be layered abstractly. Data computation of higher layer could be provided by using computations provided by the lower layers. Also, we noticed that different service features might use different computational component layers. For example, CC required the vision data to be processed at the "Object Recognition" layer, while TP required the output from the "Image Format Conversion" layer (see the Vision Manager component in Figure 4).

Based on the two observations on the data plane, we applied the third engineering principle. The resulting architecture is shown at Figure 4. Vision Manager and Audio Manager consist of layers for data computations and a controller located at the top of the layers, called Quality of Service (QoS) Manager. QoS Manager determines the level at which computation should be performed for services.

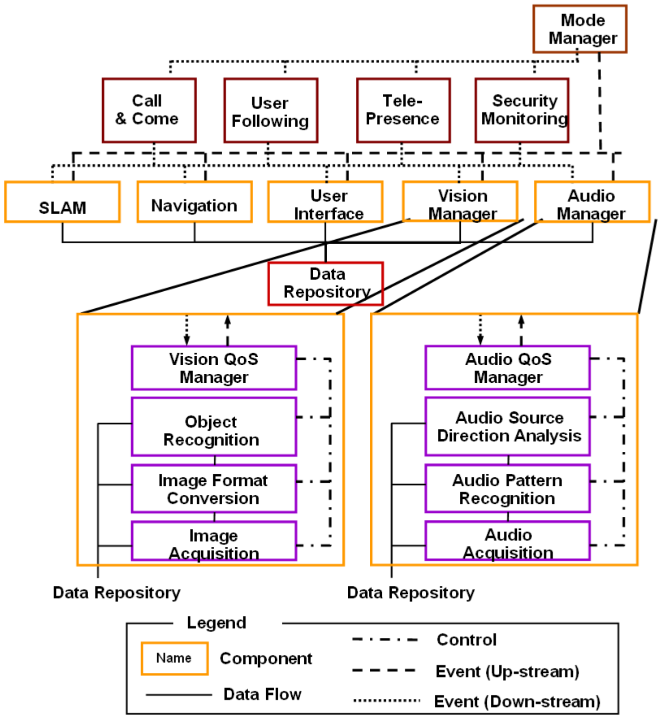


Figure 4: New SA of SHR100

4. EXPERIMENTAL RESULTS

In this section, we explain result of the re-engineering. First, we show a new SA in Section 4.1. Then, details about the control plane and the data plane are described in Section 4.2 and Section 4.3 respectively.

4.1 New Architecture

We applied the re-engineering principles explained in Section 3 to the previous architecture design of SHR 100. The new architecture is shown in Figure 4. At first, we identified five computational components - SLAM, Navigation, User Interface, Vision Manager, and Audio Manager. After identifying and separating the computational components, we could easily identify control components (CC, UF, TP, and SM). All computational components were connected via a data connector/bus. Similarly all control components were connected via a control connector/bus. Then, Mode Manager was specified to control global behavior of the robot by receiving all up-stream events and managing the control components. Each of these control components and Mode Manager was specified as a separate module in Esterel.

4.2 Control Plane

We re-engineered core implementation of the control plane written in C/C++ into Esterel. Through re-engineering, several bugs in the implementation were found. For example, a main control procedure for the CC service was implemented in `void CCallComeDlg::ProcessState()` illustrated in Figure 5. `ProcessState()` is called periodically once in every 100 ms. Given a command, CC executes through sequential "steps" each of which corresponds to a case statement block (i.e. `case` n`: ... break;`). Each step is identified by the value of `m_order` declared at line 2. At the end of each case statement block, `m_order` is updated to determine the next step. After one step is executed, `ProcessState()` is terminated and called again after 100 ms. If a new command is given between two adjacent invocations of `ProcessState()`, the previous command is ignored and the new command is processed.

```
01:class CCallComeDlg {
02:   int m_order;
03:   ...
04:void processState() {
05:   ...
06:   switch(m_order) {
07:     case 0: STOP();
08:             m_order++;
09:             break;
10:     case 1: ROTATE();
11:             m_order++;
12:             break;
13:     case 2: static int nCount = 0;
14:             if (abs(m_bef0-cur0)==0) nCount++;
15:             else nCount = 0;
16:             if (nCount > 2) m_order++;
17:             break;
18:     ...
19:     case 9: CALL_N_COME_FINISHED();
20:             m_order = -1;
21:             break;
22:   }
23:}
```

Figure 5: A main control procedure for the CC service in C++

This pattern of reactive programming is a straight-forward way to implement preemption in C/C++, but error prone. For example, at line 16, `nCount` is used for testing *two* times whether SHR stops rotation. Testing may, however, happen only *one* time. `nCount` can be greater than two all the time because `nCount` is declared as a static local variable at line 13. This error decreases the accuracy of user recognition due to blurred image captured while the robot does not stop rotation completely.

Esterel prevents such errors by handling a preemptive event e with preemption operator such as `EVERY` e `DO` *statements* `END EVERY` (see line 14 to line 28 in Figure 6). Figure 6 is a skeleton of re-implemented core of the control plane in Esterel. [2]

```
01:module control_plane:  % Control Plane
02:input EVENT: integer;
03:output STOP,ROT,GO,CC_DONE,CS_DONE,DET,N_DET;
04:signal CALL_COME, CALL_STOP in
05:run mode_man||run cnc||run uf||run tp||run sm;
06:end signal
07:end module
08:
09:module cnc:             % Call and Come service
10:function human_in_range() : boolean;
11:input CALL_COME,CALL_STOP; %come,stop commands
12:output STOP,ROT,GO,CC_DONE,CS_DONE,DET,N_DET;
13:var mv:=false:boolean,n:integer in
14:  every immediate [CALL_COME or CALL_STOP ] do
15:      present
16:      case CALL_COME do   % come command
17:        mv := true;
18:        emit STOP; pause;
19:        run rot_det;
20:        ...
21:        emit CC_DONE;pause;
22:      case CALL_STOP do   % stop command
23:        emit STOP;
24:        if mv=true then emit CS_DONE;
25:        else mv:=true;pause;run rot_det end if;
26:      end present;
27:    mv := false;
28:  end every
29:end var
30:end module
31:...
```

Figure 6: Skeleton Esterel code for the control plane

A module `control_plane` (line 1 to line 7) represents the control plane containing Mode Manager `mode_man`, the CC service `cnc`, the UF service `uf`, the TP service `tp`, and the SM service `sm`. Line 5 executes these five components concurrently using a module execution operator `run` and a parallel operator `||`. Communication among the control components (services) is implemented using *(valued) events* declared from line 2 to line 4. `mode_man` coordinates control components through these events.

A module `cnc` is defined from line 9 to line 30. At line 10, CC declares an external C function `human_in_range()` which detects if a user is within 1 meter range. Line 14 to line 28 execute when a "come" command (`CALL_COME`) or a "stop" command (`CALL_STOP`) is given. A "come" command is handled from line 16 to line 21. A "stop" command is handled from line 22 to line 25. A task of rotating the robot toward a user's direction and recognizing the user is implemented in a submodule `rot_det` (not shown in Figure 6). `rot_det` is executed for both "come" and "stop" commands at line 19 and line 25.

4.3 Data Plane

The data plane consists of five computational components (SLAM, Navigation, User Interface, Vision Manager, and Audio Manager) and one data repository. The computational components read input data from the sensors and process them to generate outputs, such as events or temporary data. The events are first sent to Mode Manager to determine the global state of the robot, and then delivered to the relevant Service Manager. The temporary data are stored at Data Repository and used as inputs for other computational components. For example, Vision Manager generates current user's location in every 200ms during the UF service. Then, Navigation determines the robot's next destination based on the user's location data in the repository.

Raw data obtained by vision or audio sensors are processed through data refinement hierarchy (layers). The layers in the computational components are identified and organized based on the hierarchy as suggested in Section 3.3. For example, as illustrated in Figure 7, image data from the front camera are first captured (L1:Image Acquisition Layer), then converted into a file format (L2:Image Conversion Layer), and finally a human face is recognized by analyzing colors in the file (L3:Object Recognition Layer).

QoS Manager controls the computational component to process data at the 'right' level. Without QoS Manager, the components always generated the most refined data whether the data was required by currently active service or not. For example, Figure 7 shows a part of behavior specifications for Vision QoS Manager. The state transitions from 'Vision Ready' to 'UF Vision' and 'CC Vision' trigger a 'Recognize Object' event, which uses service of the 'Object Recognition' layer. [3] On the other hand, TP and SM trigger a 'Convert Format' event which requires service of the 'Image Conversion' layer, not service of the 'Object Recognition' layer.

To support this hierarchy, we provided interface classes of the *Layers* architectural pattern [9] as in Figure 8. The implementation part of Figure 8 shows how the layers in Vision Manager are implemented by using the interface classes. The invocation of the `L3Svc` method in the `Vision_L3_ObjRec` class propagates down to `L1Svc` in the `Vision_L1_ImageAcq` class through the `L2Svc`, before starting its own data computation (see `if(lLayer->L2Svc())` of `L3Svc` in Figure 8).

5. LESSONS LEARNED

In this section, we summarize the lessons learned from the re-engineering project which, we believe, can be applied to other projects of similar domains.

[2] Size of the Esterel program is around 200 lines. Size of generated C code from the Esterel program is around 1000 lines. The object file compiled from this C code is 17KB.

[3] Note that the loop transition of the 'UF Vision' state means that UF requires the vision data generated in every 200 ms until the service terminates.

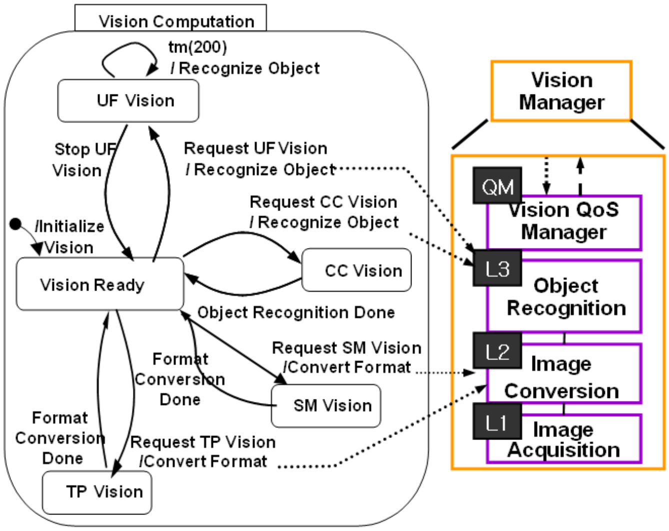
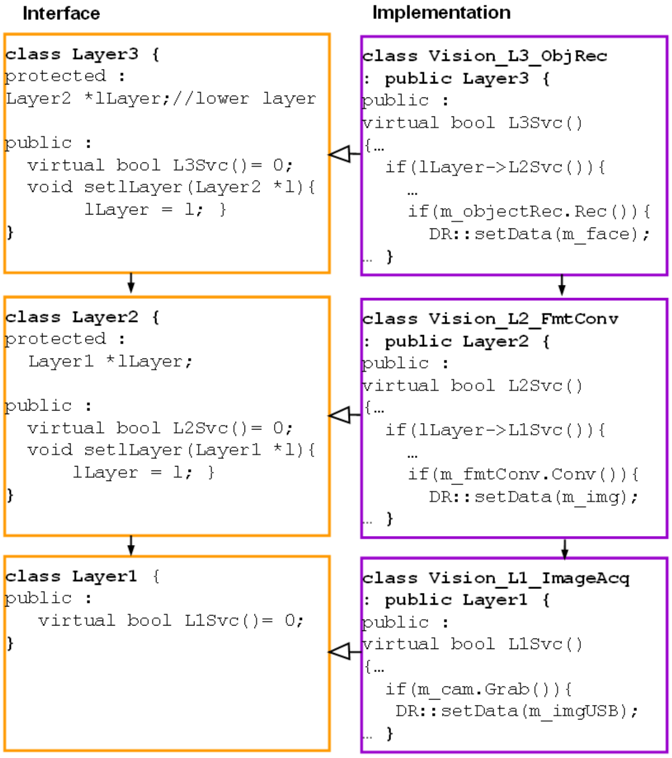


Figure 7: Vision QoS Manager



```
class Layer3 {
protected :
Layer2 *lLayer;//lower layer

public :
  virtual bool L3Svc()= 0;
  void setlLayer(Layer2 *l){
       lLayer = l; }
}
```

```
class Vision_L3_ObjRec
: public Layer3 {
public :
virtual bool L3Svc()
{…
  if(lLayer->L2Svc()){
    …
    if(m_objectRec.Rec()){
      DR::setData(m_face);
… }
```

```
class Layer2 {
protected :
  Layer1 *lLayer;

public :
  virtual bool L2Svc()= 0;
  void setlLayer(Layer1 *l){
      lLayer = l; }
}
```

```
class Vision_L2_FmtConv
: public Layer2 {
public :
virtual bool L2Svc()
{…
  if(lLayer->L1Svc()){
    …
    if(m_fmtConv.Conv()){
      DR::setData(m_img);
… }
```

```
class Layer1 {
public :
   virtual bool L1Svc()= 0;
}
```

```
class Vision_L1_ImageAcq
: public Layer1 {
public :
virtual bool L1Svc()
{…
  if(m_cam.Grab()){
   DR::setData(m_imgUSB);
… }
```

Figure 8: Layered Implementation of Vision Manager

5.1 Necessity of Re-engineering

From the experience of re-engineering SHR100, we are convinced that product re-engineering is essential, not optional in many cases. Due to limited development time and resource, developers tend to concentrate only on technology oriented components at the early stage of product development without considering how they will be integrated. Furthermore, separate team-oriented organization structure makes integration of components more difficult. Therefore, once feasibility of the project is confirmed through a prototype of a product, re-engineering the product at a later stage should be enforced for increased quality of the product. The benefits we obtained from re-engineering are as follows.

- *Convenient feature interaction analysis*
  The re-engineered SA helped analyzing feature interaction problems and finding the sources of the problems by capturing interactions among the components clearly. Without the new SA, however, analyzing such problems and tracking down their causes in the program code would be time-consuming task. It is because communication points among the components are scattered in the code and the number of the interaction points increases rapidly as the number of components increases.

- *Simple component plug-in*
  We could replace the existing UF service with a newer version with nominal code modification in the new architecture. The original UF code used only vision data from the front camera for tracking movement of the user. The newer version used vision data as well as sensor data from the structured light sensor for better accuracy of tracking the user.

- *Exhaustive code reviewing*
  Re-engineering helped to uncover subtle bugs in the original design and implementation because re-engineering required thorough code reviewing for restructuring existing implementation. For example, we detected a bug on the CC service which decreased the accuracy of detecting a user (see Section 4.2).

5.2 Separation of Priority Management

Through the project, we found that the unclear separation of global priority scheme from local ones was one of the primary causes of feature interaction problems. Before adopting the proposed software architecture, each service component was developed without considering other services. Also, how these services would be coordinated (i.e., a global service priority scheme) was determined after the components were developed. Therefore, the global priority had to be embedded later into the service components in an ad-hoc manner. As more service components were integrated, the priority often became inconsistent and unmanageable, and the robot exhibited incorrect behaviors.

For example, the priority between CC and UF was implemented inconsistently: for the structured light (SL) sensor, CC had a higher priority over UF, while UF had higher priority for controlling the actuator. This inconsistency caused the robot to behave abnormally when CC was activated during the UF service. The activated CC service made the SL sensor generate only the obstacle detection data required for CC. However, UF, which assumed the data from the SL sensor to be the footstep recognition result, continued controlling the actuator and moved the robot to unexpected directions. This example may seem trivial. But understanding such problems and tracking down their causes in program code were difficult and time-consuming tasks, as interactions between components should be analyzed thorougly.

With the new architecture, the priority scheme is separated from the service components and the manageability of priority was increased drastically because the control plane made the priority scheme visible. Also, the engineers, who developed the technology oriented components, could focus more on the computational aspects such as new algorithms for performance improvement, as long as they abide by the architecture.

5.3 Needs of Monitoring Capability

A monitoring capability is an important aid for tracking down possible sources of a problem. Determining where to put monitoring points in a system, however, can be difficult, if the role of each component and the way they interact each other are not clearly defined. The new SA that we proposed could alleviate this difficulty with clear separation of control and data planes and the distinction between control and data flows.

Suppose, for example, that the robot does not follow a user while the robot is in the UF mode. In this case, we have to figure out whether it is due to failure in recognizing the user, or controller's failure to give a move signal to the actuator. The separated control and data planes allowed us to monitor information flows conveniently by tapping the control bus and the data bus. First, we could obtain information on data plane. More specifically, we chose to monitor information obtained from the structured light sensor and the front camera. The left window in Figure 9 shows candidate positions for the user detected by the structured light sensor. The snapshot located in the right window of Figure 9 shows the user recognized from the front camera. By analyzing monitored information from these two viewpoints, we could figure out whether user recognition failed or not. Second, if user recognition is correct, we need to monitor the control plane. Control flow in the control plane is observed and displayed in other window (not shown in Figure 9).

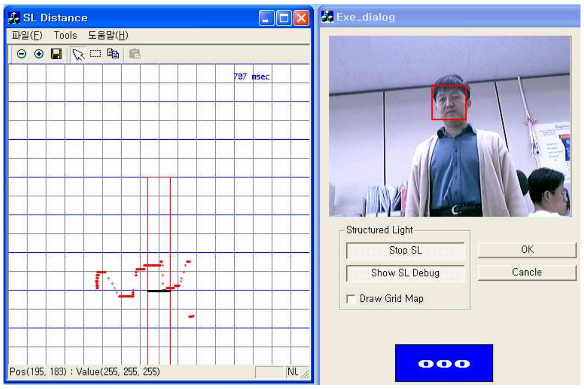


Figure 9: Monitor for the UF service

We used this monitor to figure out possible causes of misbehavior from various viewpoints. We believe that the capability of monitoring from various viewpoints is essential in developing reactive systems.

5.4 Advantages of a Reactive Programming Language

We found that a language that provides primitives for modeling reactive systems and allows refinement of the model to implementation was quite useful in the following ways:

- *Clear interactions among components*
  Esterel allows programming of interactions among the components using explicit communication mechanisms such as input/output events and signals (see Section 4.2). Events and signals are declared clearly in component definitions (modules in Esterel) and sent/received explicitly using operators such as `EMIT`/`PRESENT`. Thus, behaviors of components are easy to follow.

- *Compact implementation*
  Esterel provides pre-defined operators for expressing activities of reactive systems (e.g. communication (`EMIT` and `PRESENT`), concurrency (`||`), and preemption(`EVERY`)) which often have complex implementation when general high-level languages are used. Thus, a reactive system implemented in Esterel becomes compact and easy to analyze and update.

- *Formal analysis capability*
  An Esterel program has its formal semantics as a finite state machine (FSM), which allows rigorous analysis. Based on the FSM, a user can simulate an Esterel program step by step using the `xes` graphical simulator. Furthermore, using the `xeve` model checker [3], subtle errors (hard to detect using simulation) can be detected by exploring the whole state space. For example, we could detect and fix a feature interaction problem which caused the robot not to stop when the user gave a "stop" command. For more details about the verification result, see [17].

6. CONCLUSIONS

Hardware oriented or technology oriented organizations often consider software development as a last-minute task that can be achieved by simply integrating technology intensive components in a bottom-up way. In most cases, however, the components have to interact with each other to provide services that customers demand and their feature interactions often cause a system failure or malfunction. Therefore, we need a sound software engineering approach that supports both top-down and bottom-up views to coordinate the integrated components correctly.

We have reported a case study of re-engineering a home service robot SHR. First, we propose three engineering principles to design a SA of SHR100 - separation of control plane from data plane, hierarchy of global behavior and local behavior, and layering of data manipulation components. Then, according to these principles, we designed a new SA of SHR100 and re-engineered existing source code to conform to the SA. We used Esterel for re-engineering the control plane of SHR100 to take advantage of its reactive operators as well as clear compositionality. By this re-engineering, interactions among the components became visible and the responsibility of behaviors could be assigned to components clearly. We could detect and solve a feature interaction problem which caused the robot not to stop when a user gave a "stop" command.

As a future work, we will handle the resource management problem which is frequent source of unstable behaviors such as stuttering movement and ignorance of user input under heavy resource utilization. For this purpose, Monitoring and Checking framework [18] can be explored. Furthermore, we are considering a real-world virtual prototyping framework such as ASADAL/OBJ [20] to reduce development time and cost [16].

7. REFERENCES

[1] Honda asimo home page. http://asimo.honda.com/.

[2] Sony aibo home page. http://www.sony.net/Products/aibo/.

[3] A.Bouali. Xeve: an esterel verification environment. Technical report, INRIA, Dec. 2000.

[4] R. C. Arkin and T. R. Balch. Aura: Principles and practice in review. *Journal of Experimental and Theoretical Artificial Intelligence(JETAI)*, Volume 9(Number 2/3):175–188, April 1997.

[5] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[6] G. Berry. The foundations of esterel. *In Proof, Language and Interaction: Essays in Honour of Robin Milner,* MIT Press, 2000.

[7] J. Borrelly, E. Coste-Maniére, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The orccad architecture. *International Journal of Robotics Research*, 17(4):338–359, 1998.

[8] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[10] E. J. Cameron and H. Velthuijsen. Feature interactions in telecommunications systems. *IEEE Communications Magazine*, 31(8):46–51, Aug 1993.

[11] E. Coste-Manière and R. Simmons. Architecture, the backbone of robotic systems. *IEEE International Conference on Robotics and Automation*, 2000.

[12] E. Coste-Maniére and N. Turro. The maestro language and its environment : Specification, validation and control of robotic missions. *Proceedings of the 10th IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1997.

[13] A. C. Domínguez-Brito, D. Hernández-Sosa, J. Isern-González, and J. Cabrera-Gámez. Integrating robotics software. *IEEE International Conference on Robotics and Automation*, 2004.

[14] B. Espiau, K. Kapellos, and M. Jourdan. Formal verification in robotics: Why and how? *International Symposium on Robotics Research*, Oct 1995.

[15] R. N. T. et. al. A component- and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406, 1996.

[16] K. Kang, M. Kim, J. Lee, B. Kim, Y. Hong, H. Lee, and S. Bang. 3d virtual prototyping of home service robots using asadal/obj. *International Conference on Robotics and Automation*, 2005.

[17] M. Kim, K. Kang, Y. Hong, H. Lee, and S. Bang. Formal verification of the robot movements. *International Conference on Robotics and Automation*, 2005.

[18] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 2004.

[19] D. Kortenkamp and A. C. Schultz. Integrating robotics research. *Autonomous Robots*, 6:243–245, 1999.

[20] J. Lee, H. Kim, and K. Kang. A real world object modeling method for creating simulation environment of real-time systems. *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Volume 9:93–103, Oct 2000.

[21] R. Pack, D. M. Wilkes, and K. Kawamura. A software architecture for integrated service robot development. *IEEE International Conference on Systems, Man and Cybernetics*, 1997.

[22] L. Pinzon, H.-M. Hanisch, M. Jafari, and T. Boucher. A comparative study of synthesis methods for discrete event controllers. *Formal Methods in System Design*, 15(2):123–267, 1999.

[23] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[24] P. Zave. Architectural solutions to feature-interaction problems in telecommunications. *Feature Interactions in Telecommunication and Software Systems V*, Sep 1998.