

## 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]

### 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.

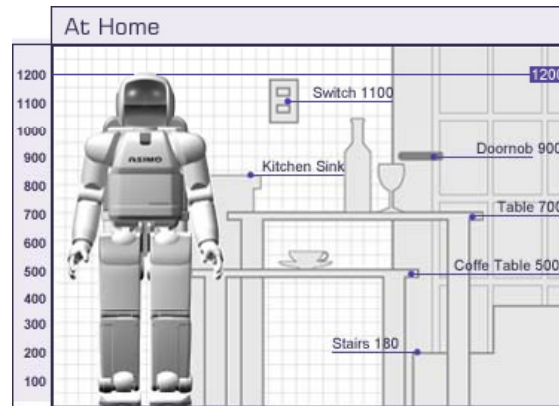
Due to limited development resources, developers of home service robots tend to focus on technology intensive components at an early stage of product development without an architectural consideration of how they will be integrated to create services. Furthermore, engineers are often grouped into separate 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 ad-hoc and bottom-up way. As a consequence, products often suffer from feature interaction problems [10, 24]. Feature interaction problems found in systems developed without

# Re-engineering Home Service Robots Improving Software Reliability: A Case Study

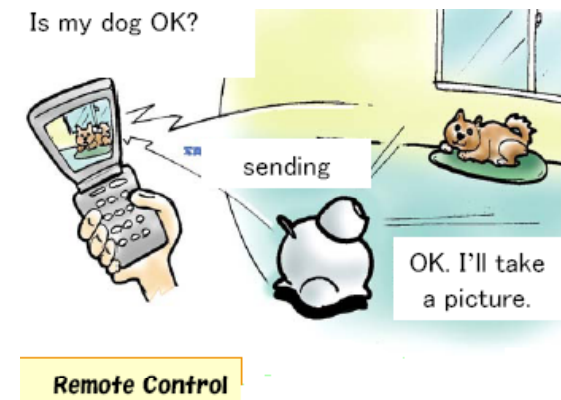
Moonzoo Kim, et. al.  
*International Conference  
on Software Engineering*  
2005

- **Introduction**
- **Re-engineering Software Architecture**
- **Control Plane Re-engineering**
- **Data Plane Re-engineering**
- **Lessons Learned**

- Designed for providing various services to human user
  - Service areas : home security, patient caring, cleaning, etc
  - Markets for home service robots are still being formed

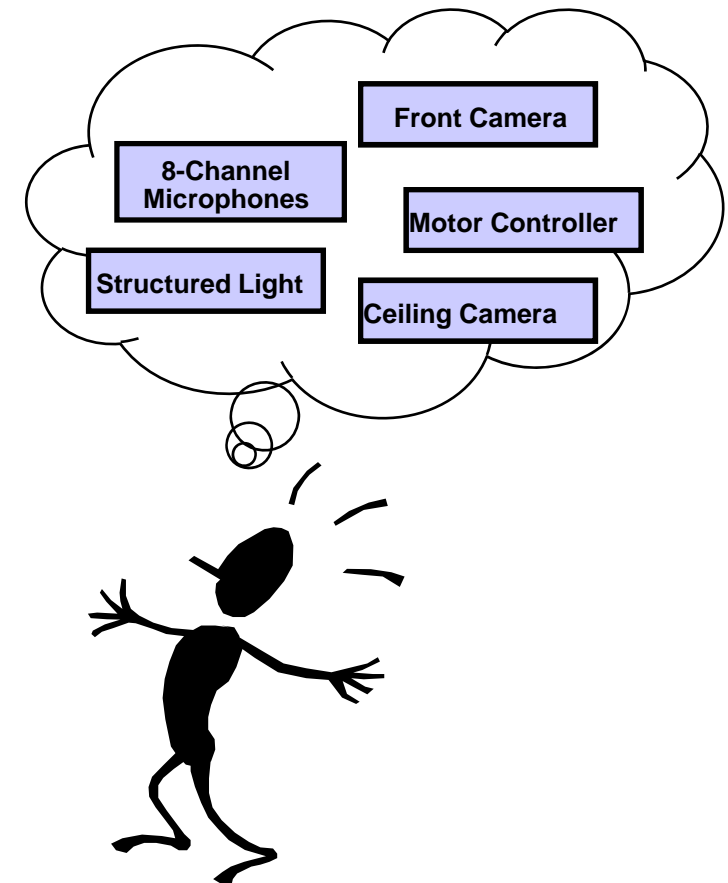
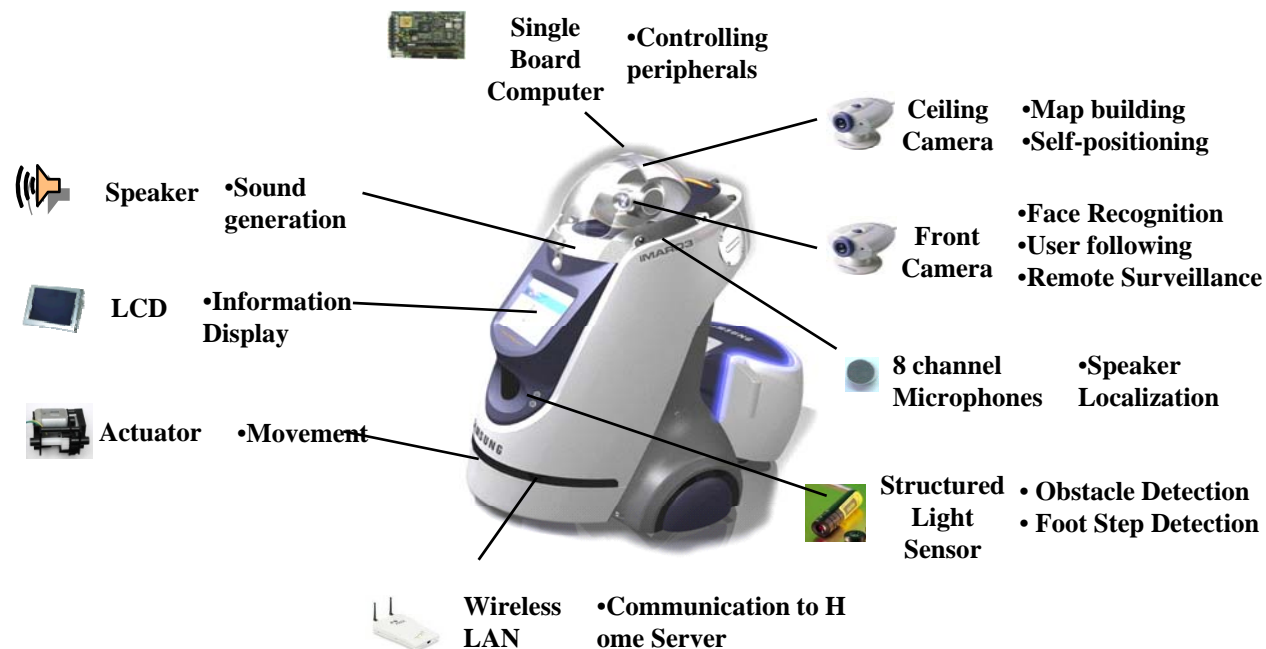


\* The above heights are examples to serve as a reference(mm).

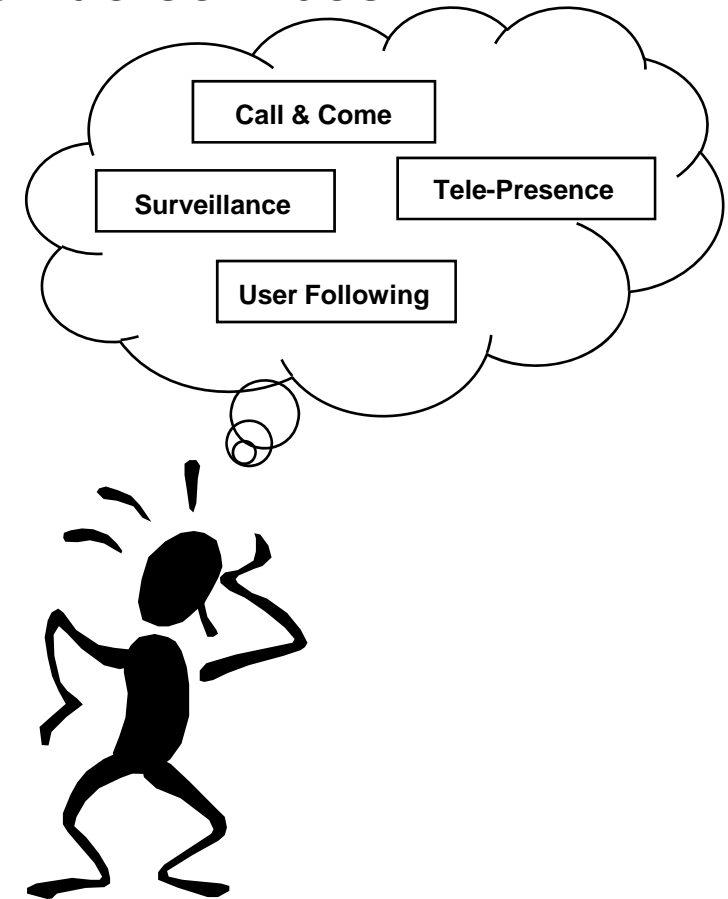
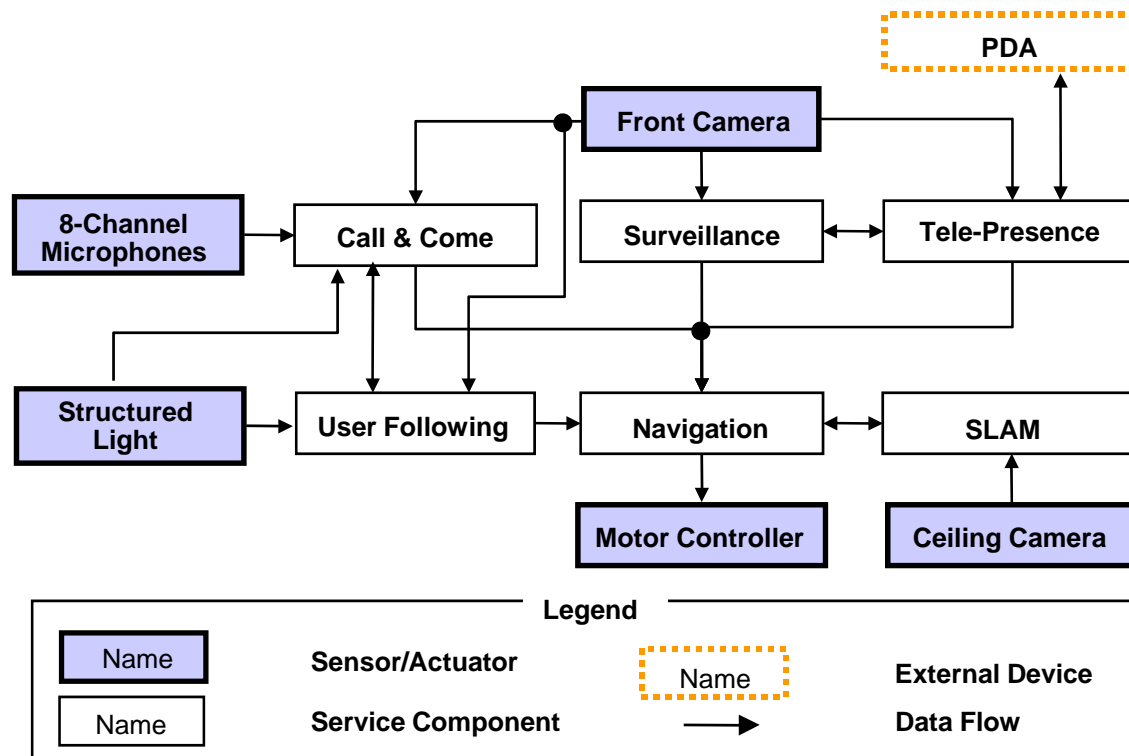


- SAIT started development of SHR00 from 2002
  - 4 separate teams (13 persons)
    - Vision recognition, speech recognition, simultaneous localization and mapping (SLAM), actuator
- Both SHR00 and SHR50 suffered feature interaction problems
  - SAIT decided to develop SHR100 from scratch
- SAIT requested POSTECH to improve the reliability of SHR100 in six months
  - SHR100 is written in 17K line of C/C++

- Robots are created based on various **technical components**
  - Speech recognizer, vision recognizer, actuator, etc



- Robot developers concentrate on technical components only, resulting in **integration in an ad-hoc and bottom-up way**
  - Difficult to coordinate components to provide services



- Problems due to bottom-up integration
  - Lack of global view
  - Difficulty in analyzing the behavior of integrated systems
  - Integration often requires modifications of other components
- Feature interaction problems
  - Invisible interactions between the components
  - Difficulty to trace the cause of problems (debugging difficulty)

*Cannot develop products in reasonable project time*



*Cannot evolve according to quickly changed market demands*

*Cannot satisfy required quality attributes (e.g. safety and temporal properties)*

- To provide **hierarchical and modular SA**
  - **Top-down** global views
  - Visualization of component **interactions**
  - High adaptability for evolving features/ technologies
- To apply **formal construction & verification** to the core of SW
  - Rigorous and **automated debugging support**
  - Explicit interaction mechanism among components
  - Compact and easy-to-understand code  
(domain specific programming language)

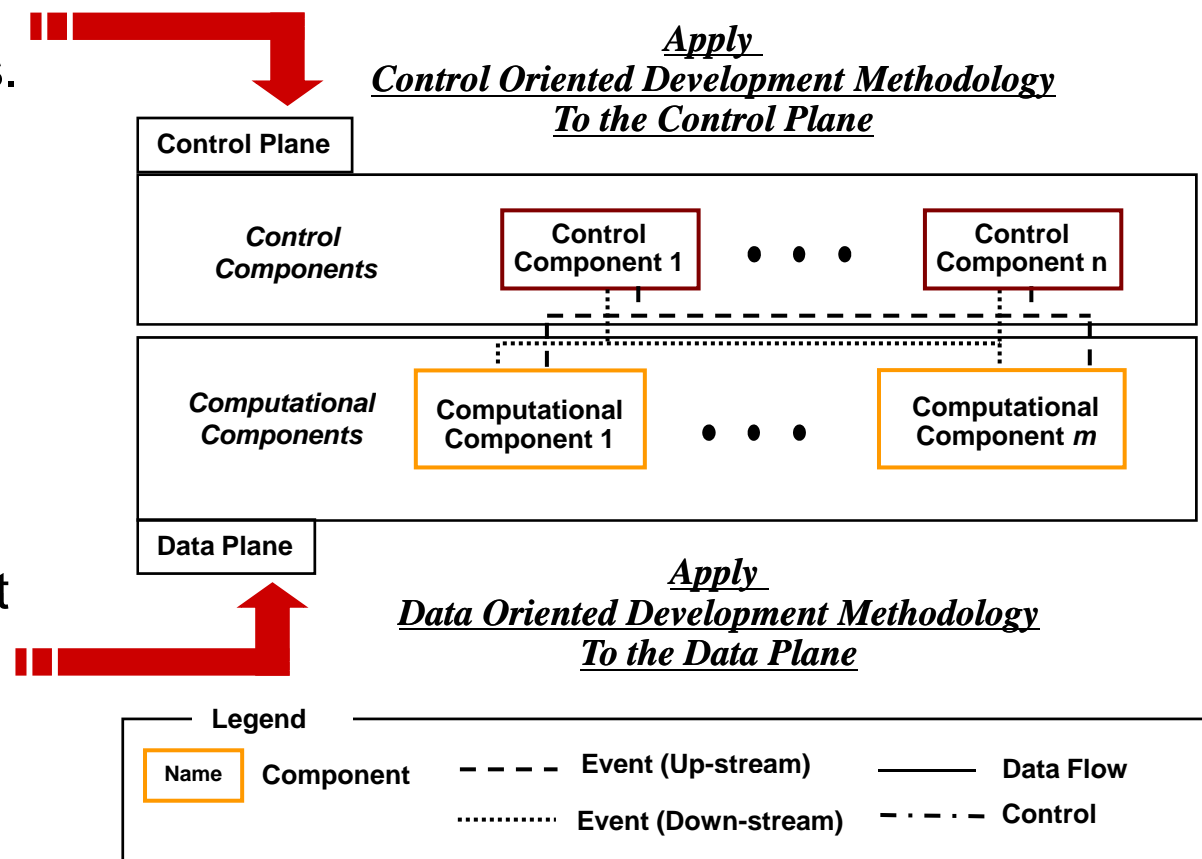


- Re-engineering based on the following three **principles**
  1. Separation of control plane from computational plane
  2. Distinction between global behavior and local behavior
  3. Layering in accordance with data refinement hierarchy

## ■ Principle1: Separation of Control Components from Computational Components.

The first class of data is **control data** for handling robot behaviors.  
: correctness is the foremost concern due to complexity of reactive system.

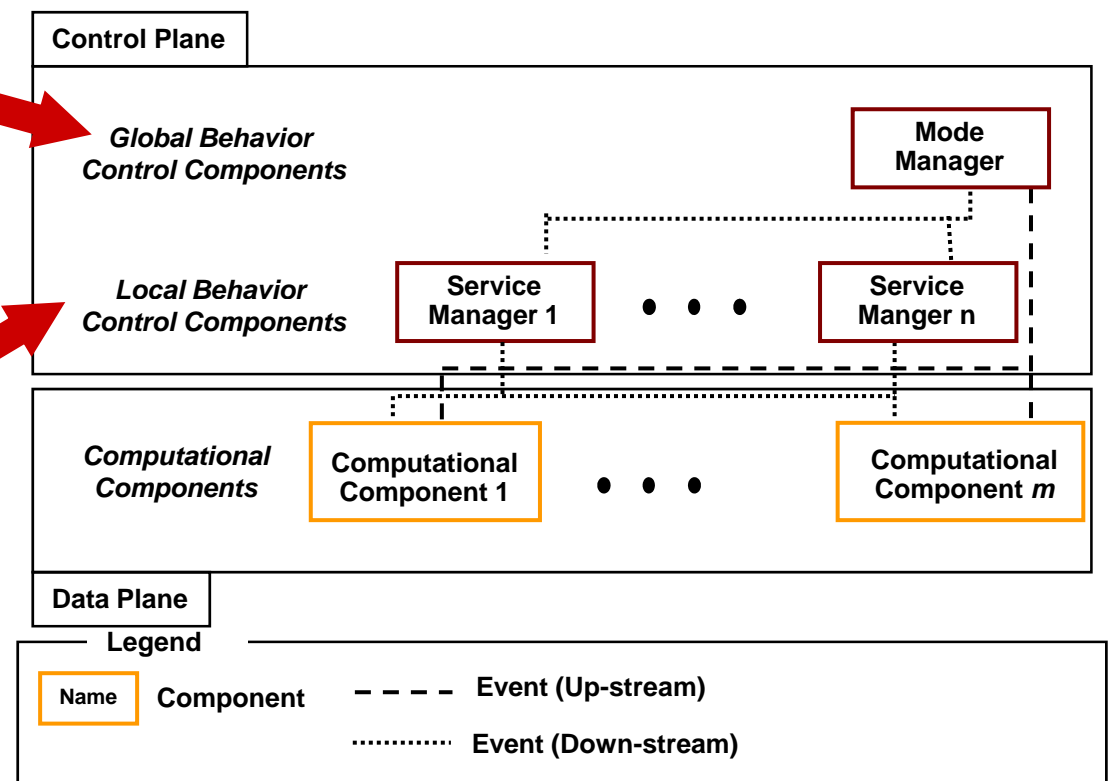
The second class of data is **computational data** for handling robot function.  
: efficient computation is the most important goal.



## ■ Principle2: Separation of Local Behaviors from Global Behaviors

**Mode manager** components defines the system modes and the interaction policy between service components.

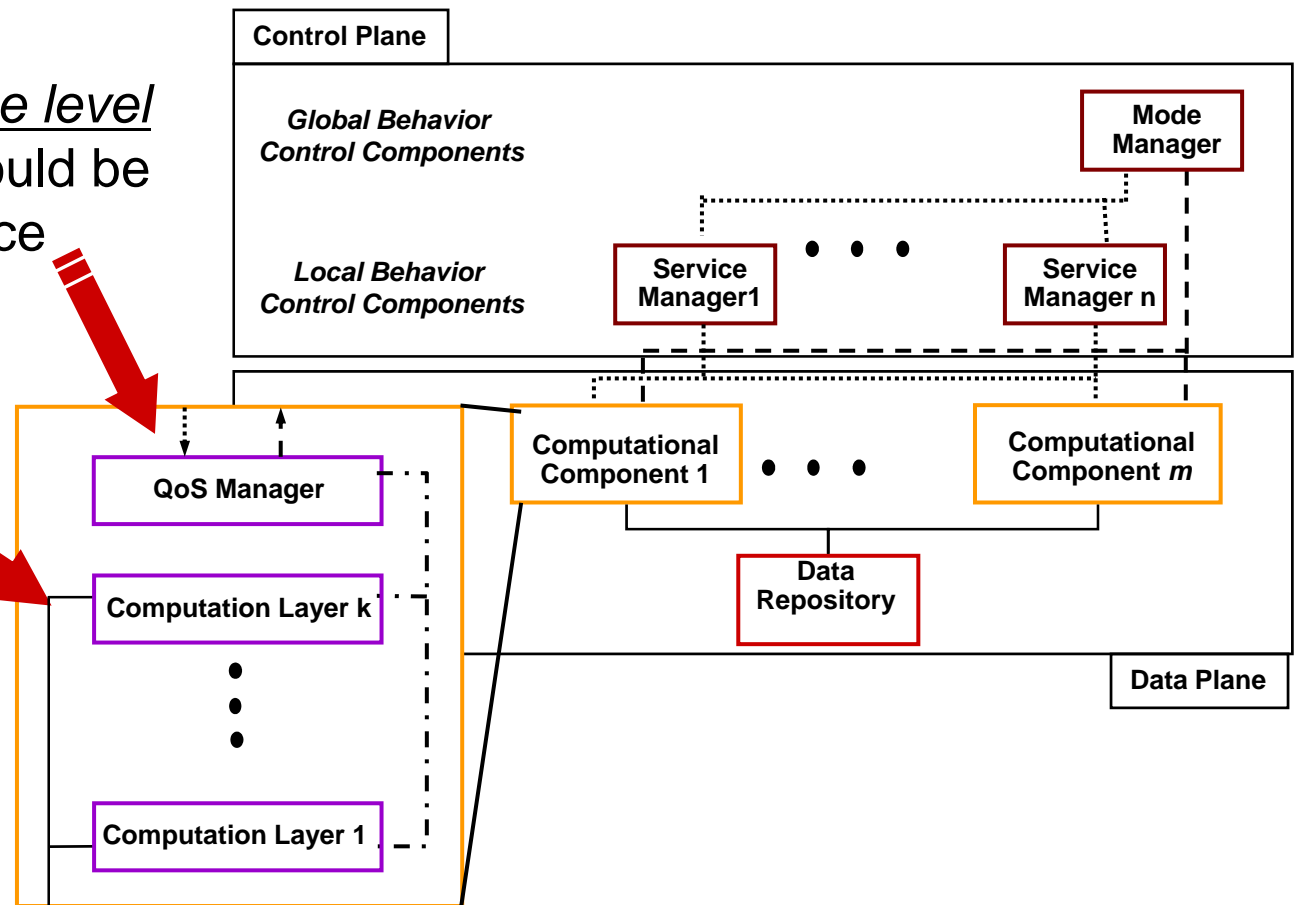
**Service manager** components defines the behavior of service feature by controlling the computational components.



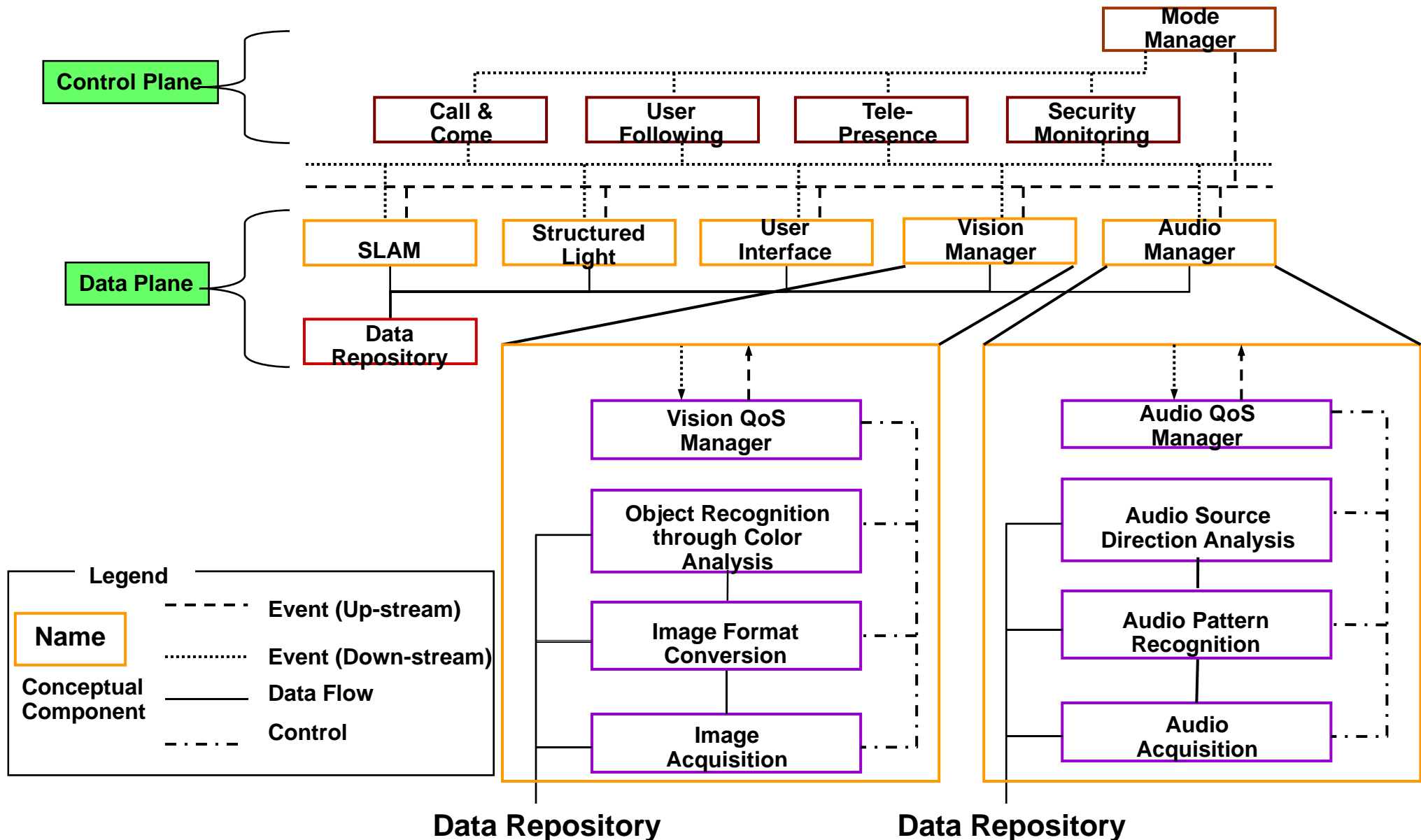
## ■ Principle3: Layering in Accordance with Data Refinement Hierarchy

**QoS Manager** determines *the level at which the computation* should be performed according to service

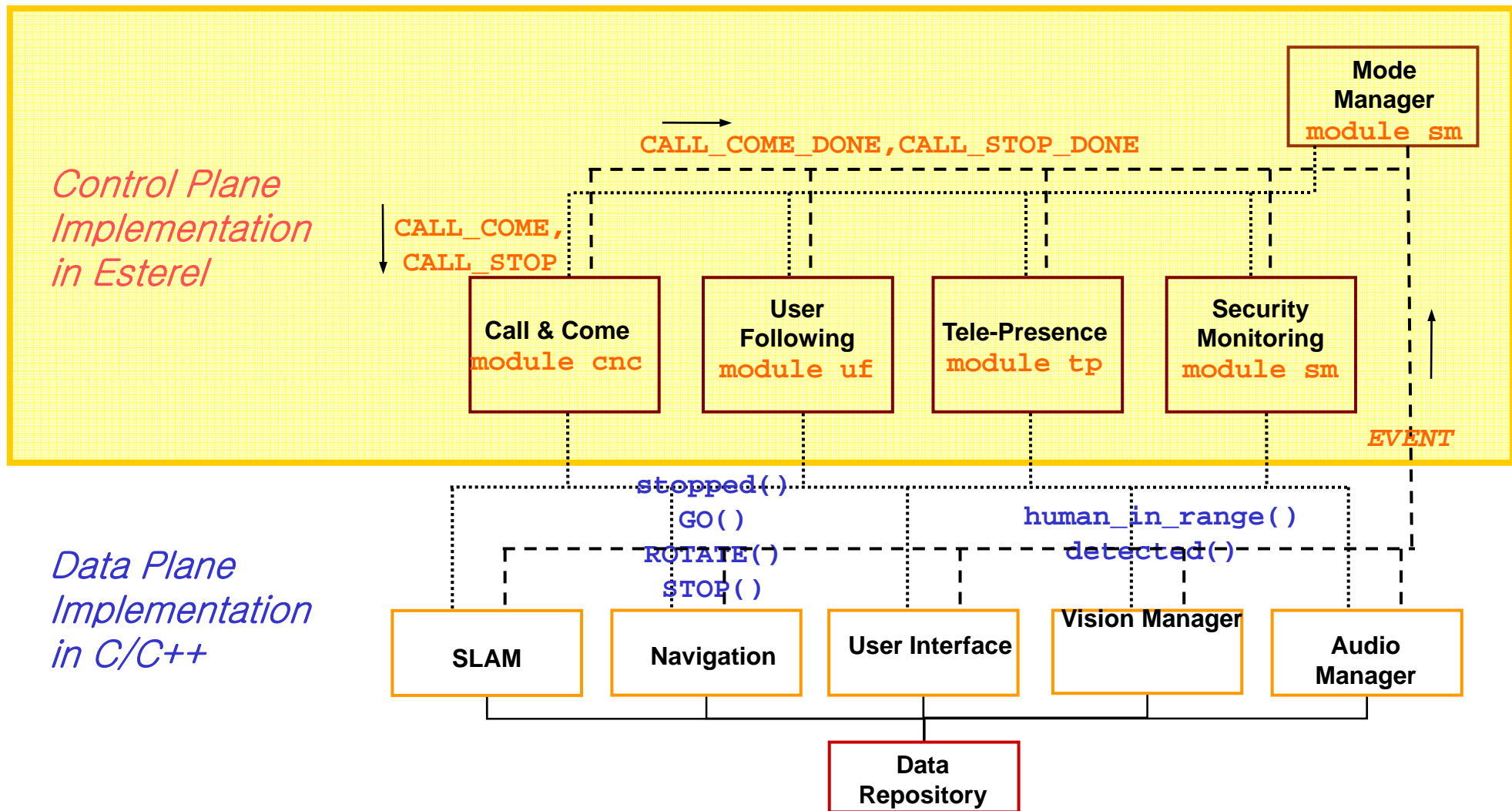
There exist **data refinement hierarchy** for data computation and different service features may use *different computational component layers*.



# New Software Architecture *Re-engineering Software Architecture*



# Re-engineering Control Plane (1/3) *Re-engineered SHR100 Architecture*



- A main control procedure for the *preemptive* CC service

*New Commands*

```
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_berO-curO)==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:         } /* End of processState()
23:     }
```

- processState() is called periodically once in every 100 milliseconds.
- CC executes through sequential steps identified by the value of m\_order
- nCount is declared as a static local variable at line 13

- This straightforward pattern is *error prone*.

## Re-engineering Control Plane (3/3)

## Overview of the re-engineered CC Implementation

```
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:...
```

- Esterel handles a preemptive event *e* with a preemption operator

*EVERY e DO statements  
END EVERY.*

- Interactions among Esterel modules are clearly defined via events

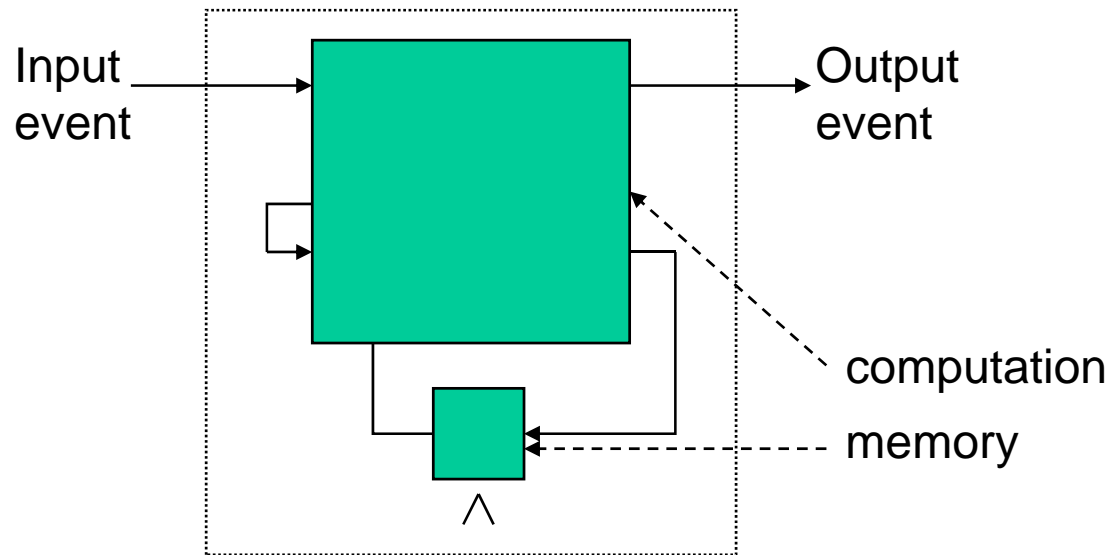
*PRESENT CASE e DO  
statements END PRESENT*

- Submodule can be conveniently utilized

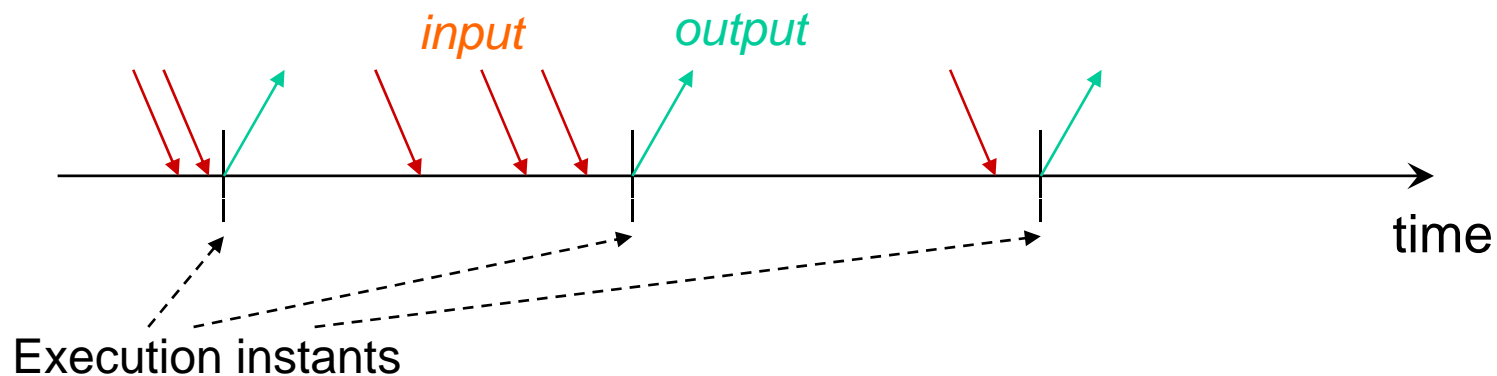
*RUN module*



# Esterel Background (1/5) *Reactive Synchronous Language Esterel*



- Synchrony = abstraction of the real world
- Cycle-based execution model, global clock
- Perfect synchrony

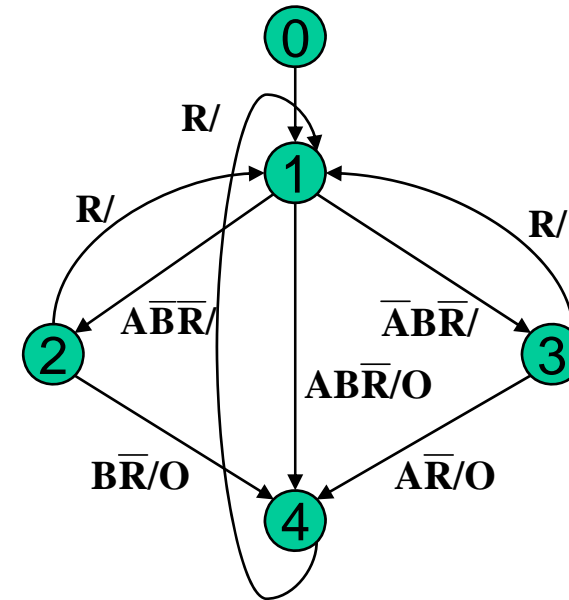


- Synchronous language
- Structural imperative style
- Basic constructs
  - Classical control flow  
 $p; q, p \mid q, \text{loop } p \text{ end}$
  - Signals:  
 $\text{signal } S \text{ in } p \text{ end, emit } S,$   
 $\text{present } S \text{ then } p \text{ else } q \text{ end}$
  - Preemption  
 $\text{abort } p \text{ when } S, \text{ every } s \text{ do } p \text{ end every}$
  - Exception handling  
 $\text{trap } T \text{ in } p \text{ end, exit } T$

### ABRO example

```

Input A, B, R;
Output 0;
loop
[
  await A
  ||
  await B
];
emit 0;
halt
every R
    
```



```

switch(state){
case 0: state=1; break;
case 1: if(!R)if(A)if(B) {0(); state=4; }
        else state=2;
        else if(B) state=3; break;
case 2: if(R) state=1;
        else if(B) {0(); state=4; } break;
case 3: if(R) state=1;
        else if(A) {0(); state=4; } break;
case 4: if(R) state=1; break;
}
    
```

### The esterel Compiler:

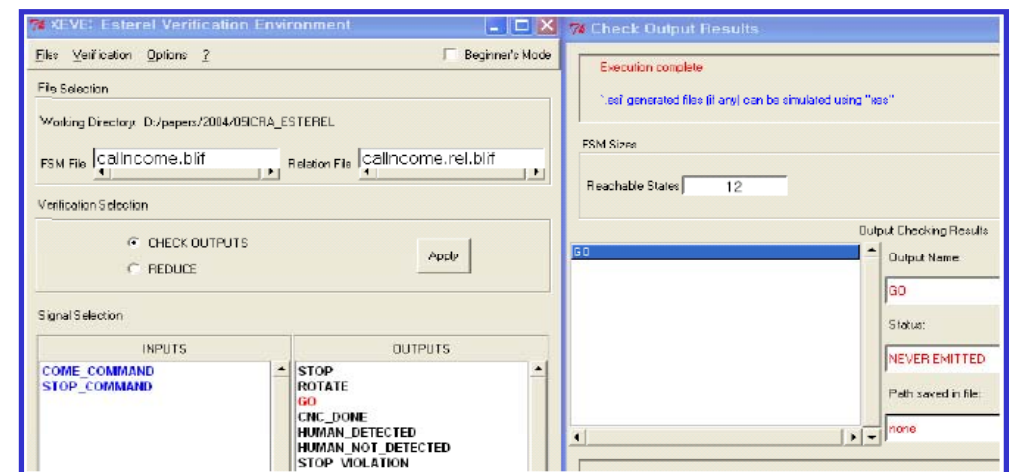
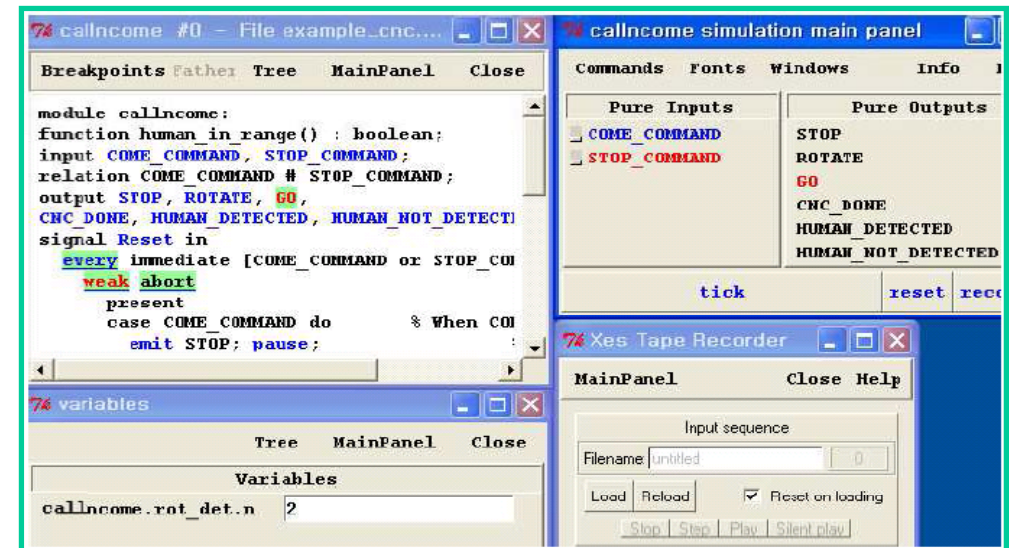
- C/VHDL/Verilog code generation.
- interface between Esterel and C.

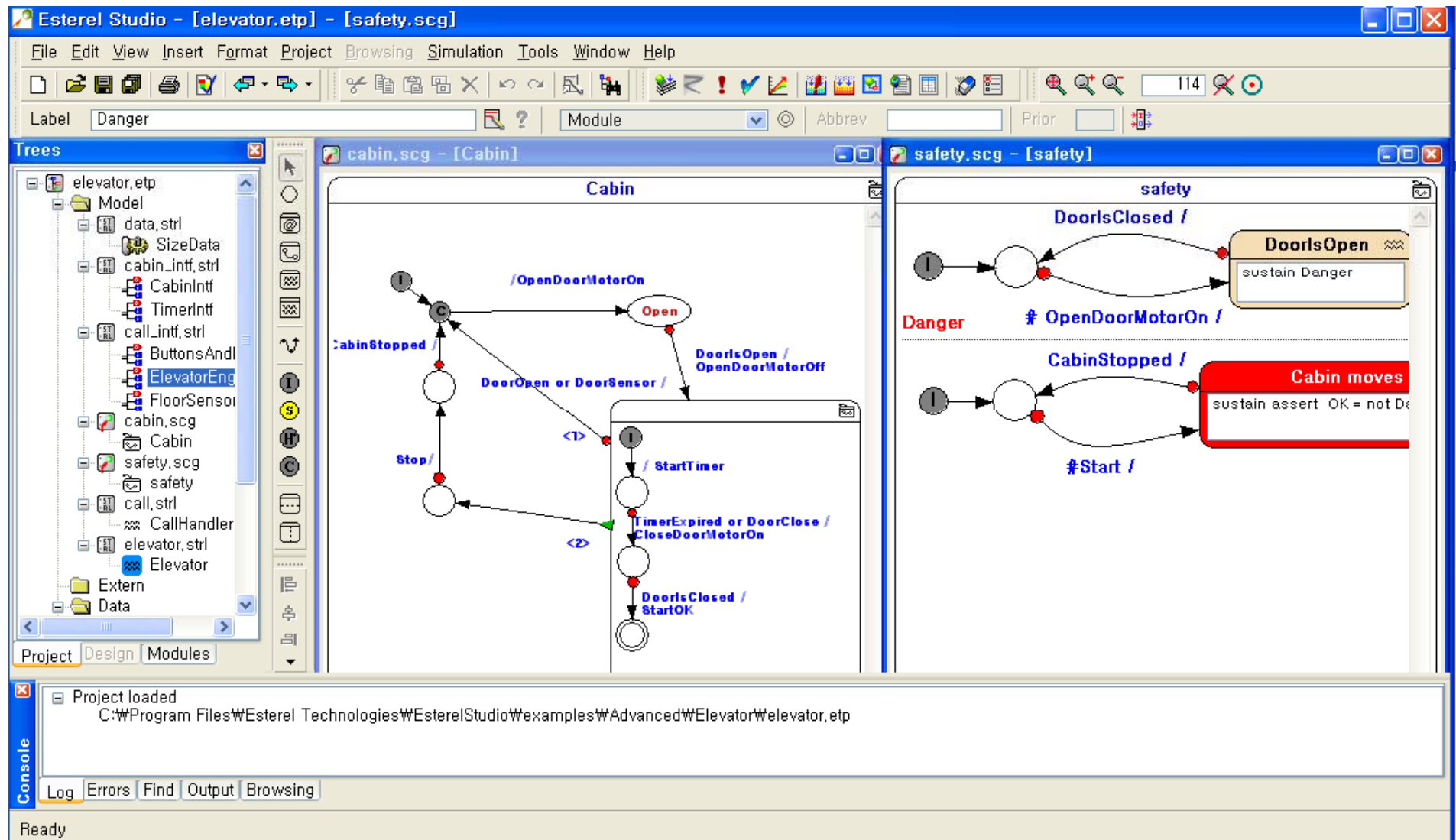
### The xes Graphical Simulator:

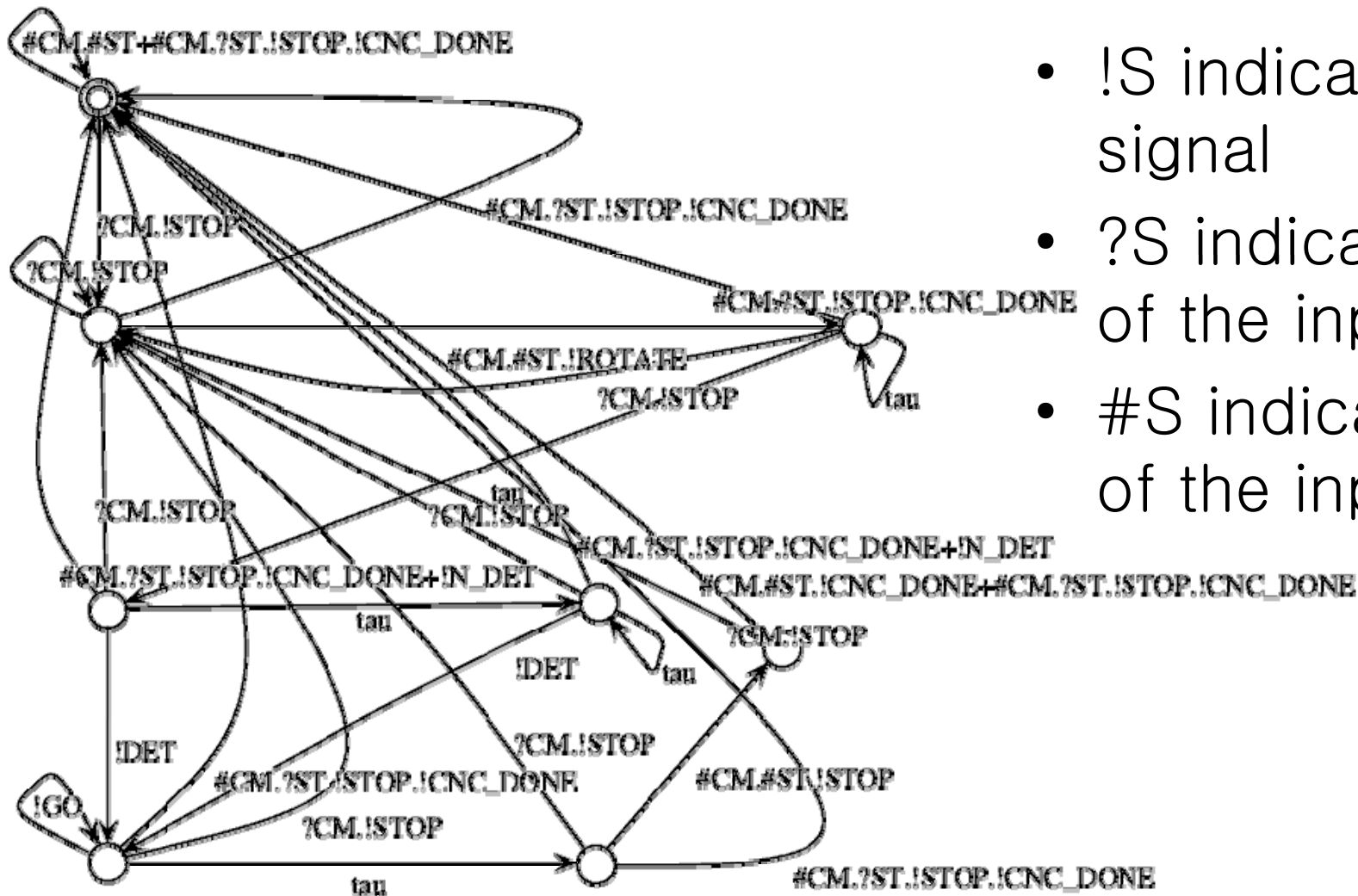
- graphical interactive simulation
- session recording/replay.

### The xeve Model Checker:

- analyzes an Esterel program.
- check presence of an output signal with given configuration of input signals.







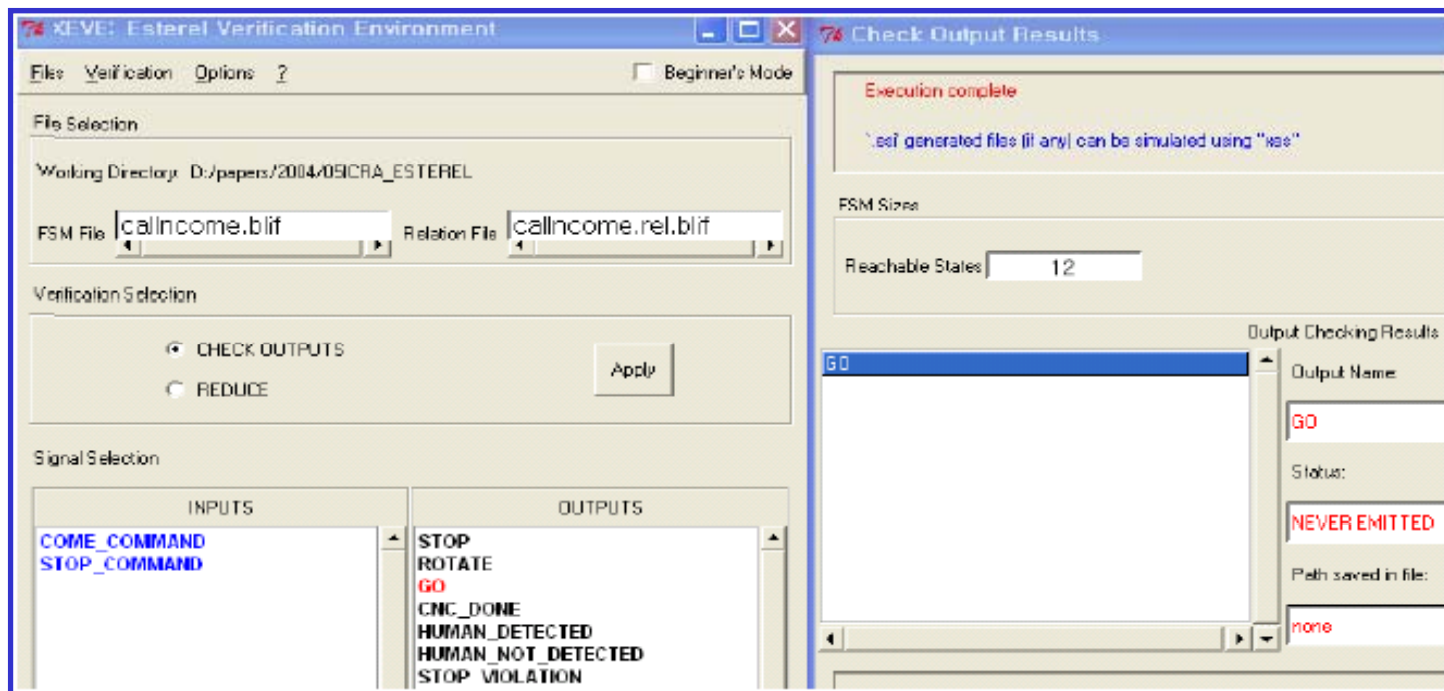
- !S indicates output signal
- ?S indicates presence of the input signal S
- #S indicates absence of the input signal S

- Stopping behaviors are *safety critical*
- Three properties on the stopping behaviors
  - P1: If a user does not give a command to the robot, the robot must not move.
  - P2: If a user does not give a “come” command, but may give a “stop” command to the robot, the robot must not move.
  - P3: If a user gives a “stop” command, the robot must stop and not move without any new command.
- We verify whether P1,P2, and P3 are satisfied in the following two cases
  - When the CC service runs solely
  - When the CC service and the UF service run concurrently



## Formal Verification of Stopping Behaviors (3/5) *Verification Result I*

- We check **P1** by setting
  - Input signals COME\_COMMAND and STOP\_COMMAND as “always absent”
  - Output signal GO to check.
- Both cases satisfy **P1**





## Formal Verification of Stopping Behaviors (4/5) *Verification Result II*

---

- The CC service satisfies *P2*, but *not CC and UF together*.
  - Verification result said that *ROTATE* and *GO* could be possibly *emitted* when COME\_COMMAND command was absent and STOP\_COMMAND might be given
  - I.e. *feature interaction* happens
- UF should have been triggered *only* after a “come” command
  1. We refined CNC\_DONE into CNC\_COME\_DONE and CNC\_STOP\_DONE.
  2. We modified the UF implementation so that only CNC\_COME\_DONE could invoke UF.
  3. After this modification, we could see that *P2 was satisfied by the concurrent CC and UF services*.

## Formal Verification of Stopping Behaviors (5/5) *Verification Result III*

### ■ The property P3.

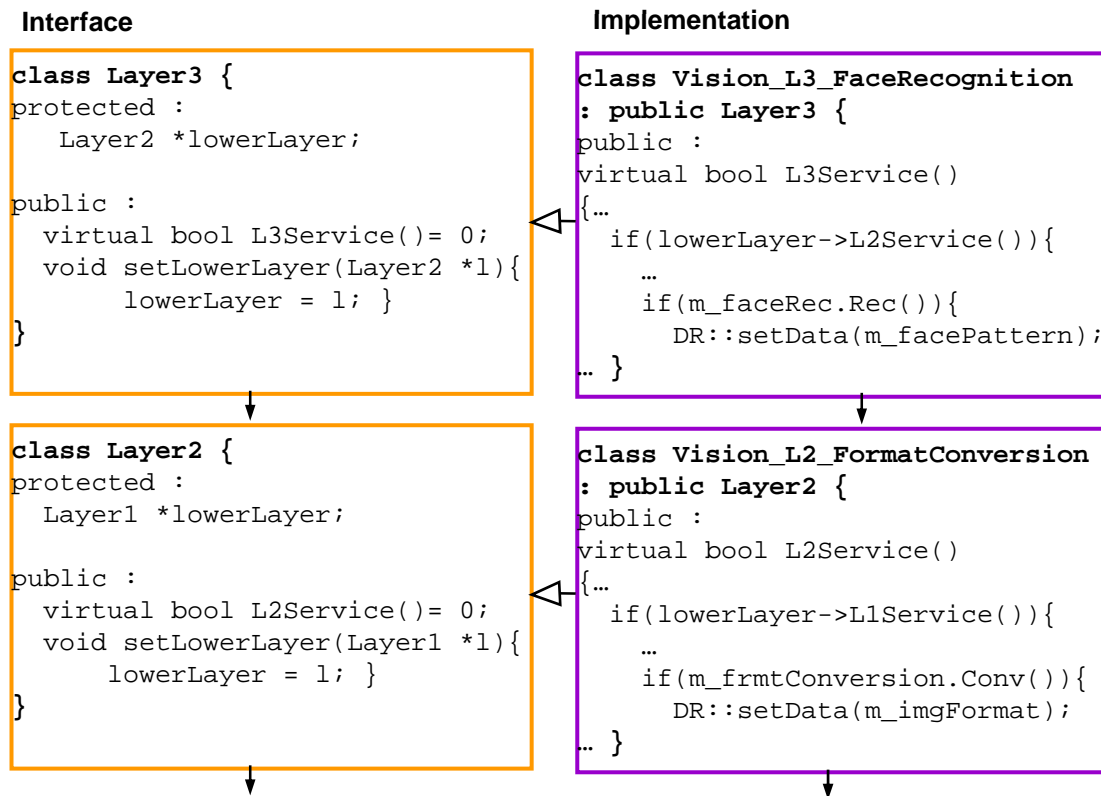
- P3: If a user gives a “stop” command, the robot stops and does not move without any new command.

### ■ To verify P3, we need to build an *observer* to detect violations

```
01:module observer:
02:input  STOP_COMMAND, COME_COMMAND, ROTATE, STOP, GO;
03:output STOP_VIOLATION;
04:weak abort
05:  every immediate STOP_COMMAND do
06:    present STOP then
07:      loop
08:        present [ROTATE or GO]
09:          then emit STOP_VIOLATION;
10:        end present;
11:      pause;
12:    end loop;
13:  end present
14:  emit STOP_VIOLATION;
15: end every
16:when COME_COMMAND;
17:end module
```

### ■ Layered Implementation of Vision Manager

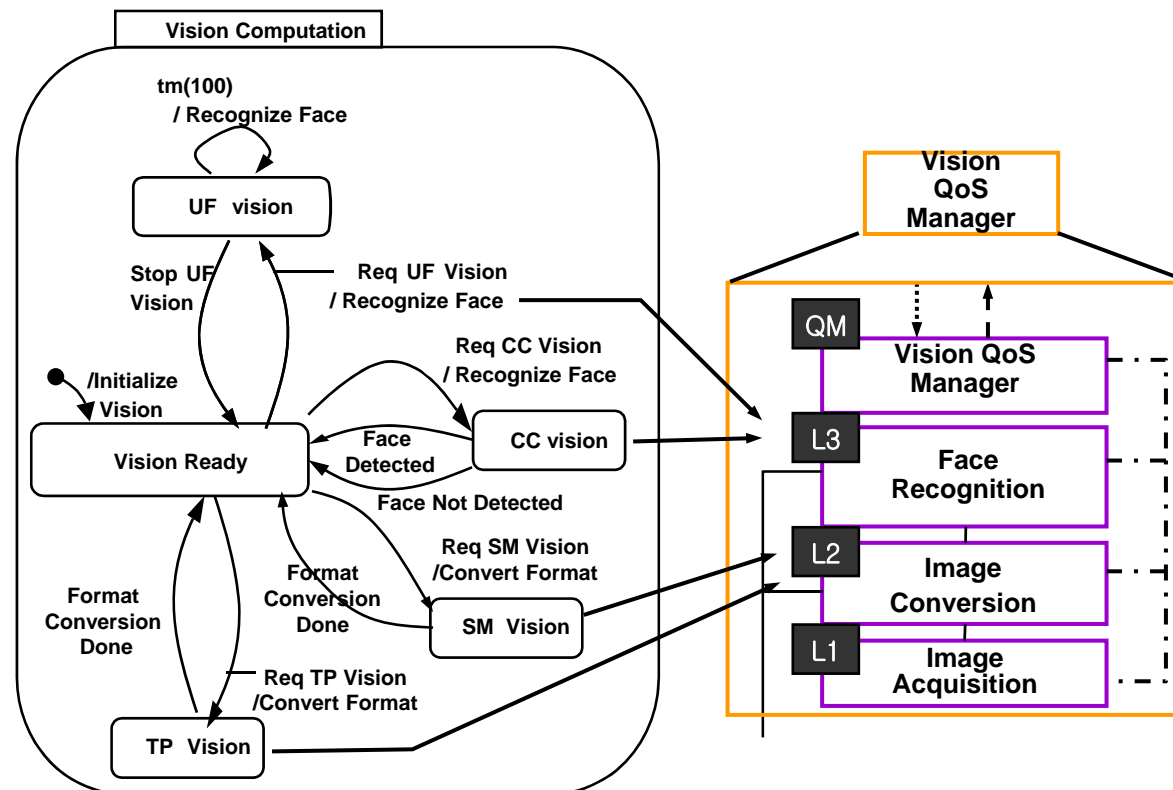
- The **layered architectural pattern** is organized based on the data refinement hierarchy.



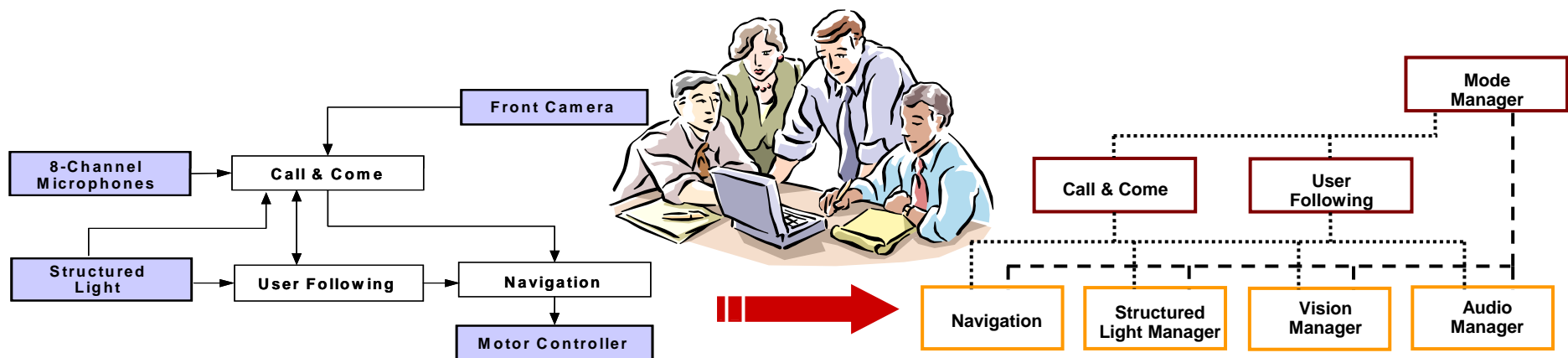
1. Image data from the front camera are captured **(Layer 1)**,
2. then converted into a file format **(Layer 2)**
3. finally a human face is identified by analyzing colors in the file **(Layer 3)**.

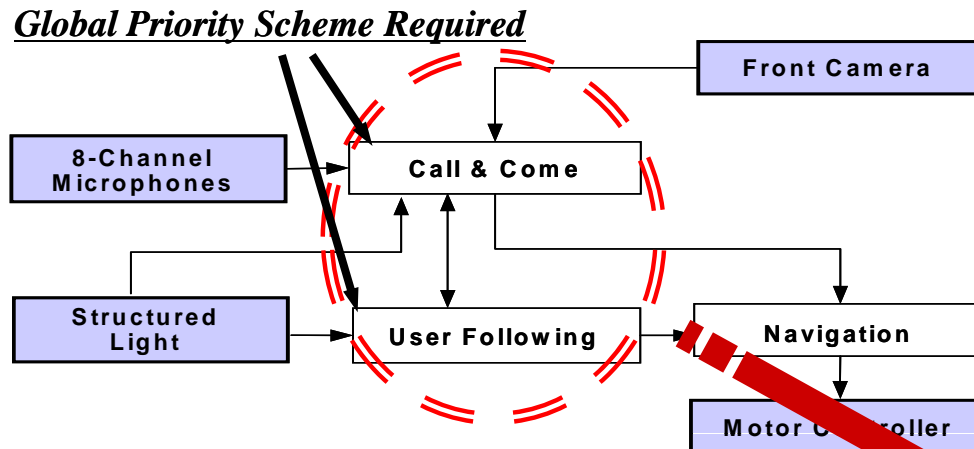
### ■ Vision QoS Manager

- The QoS manager layer selects the 'right' level of data refinements.



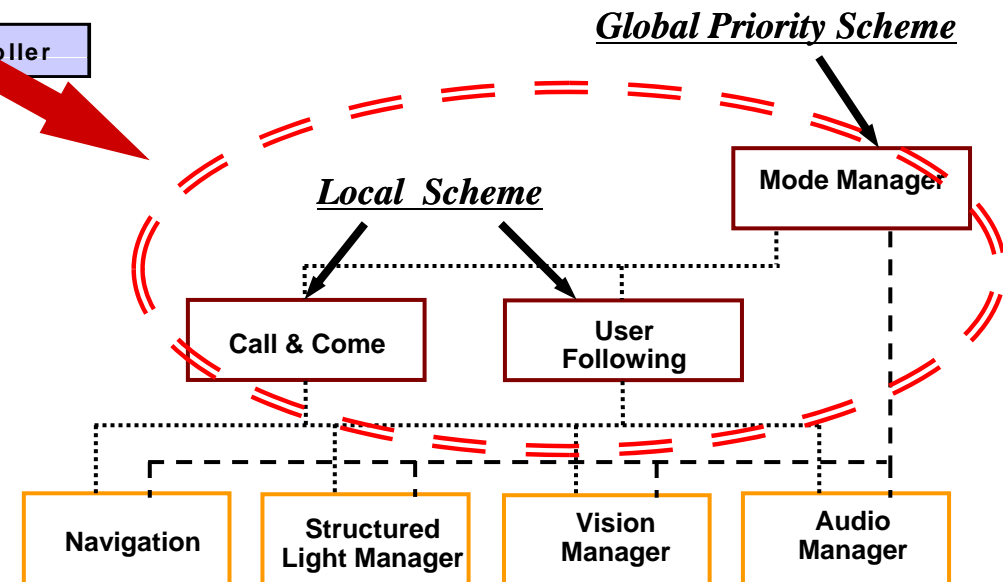
- From the experience of re-engineering SHR100, we are convinced that **re-engineering is essential**
  - Due to the limited development time, developers tend to concentrate only on **technical components at the early state** without considering how they will be integrated.
  - Once feasibility of the project is confirmed through an early prototype, **re-engineering the product at later stage** should be enforced for increased quality of the product.





-With the new architecture, the **global priority scheme is separated** from the service components and manageability of priority was increased drastically.

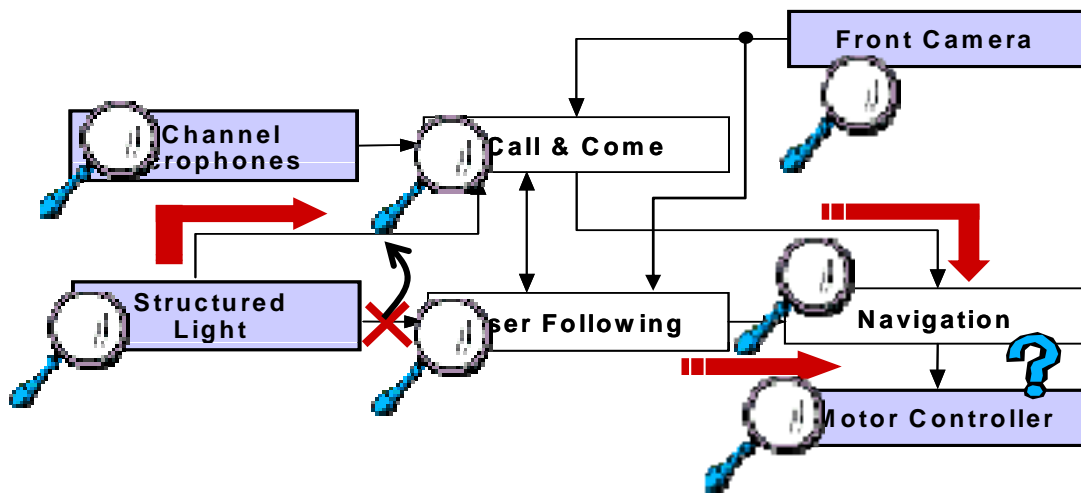
- We found that unclear global priority scheme was one of the primary causes of feature interaction problems.



# Needs of Monitoring Capability

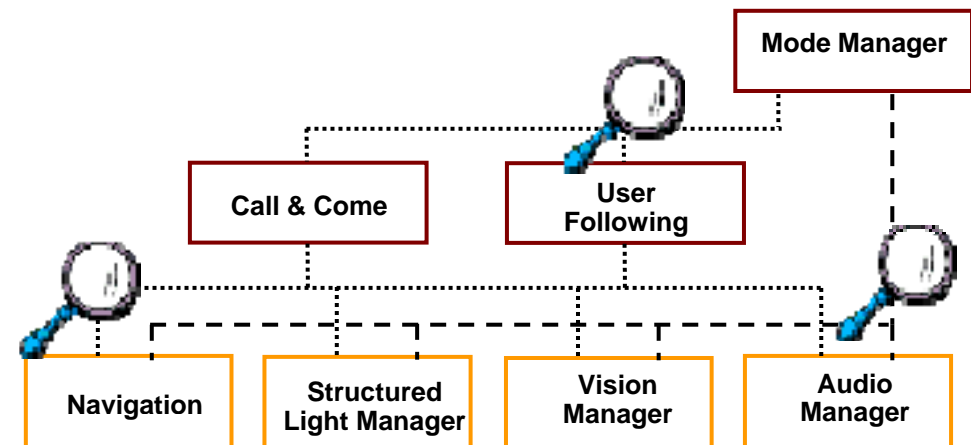
## *Lessons Learned*

- A monitoring capability is an important aid for tracking down possible sources of a problem.



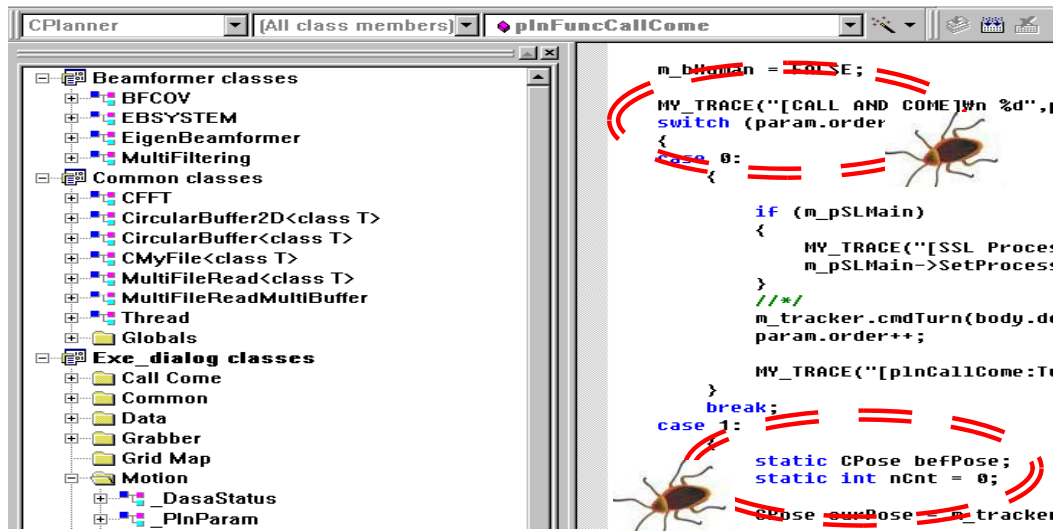
-Determining where to put probes is difficult, if the role of each component and the ways they interact each other are not clear

-The new SA that we proposed could alleviate this difficulty with **clear interaction strategy between components**



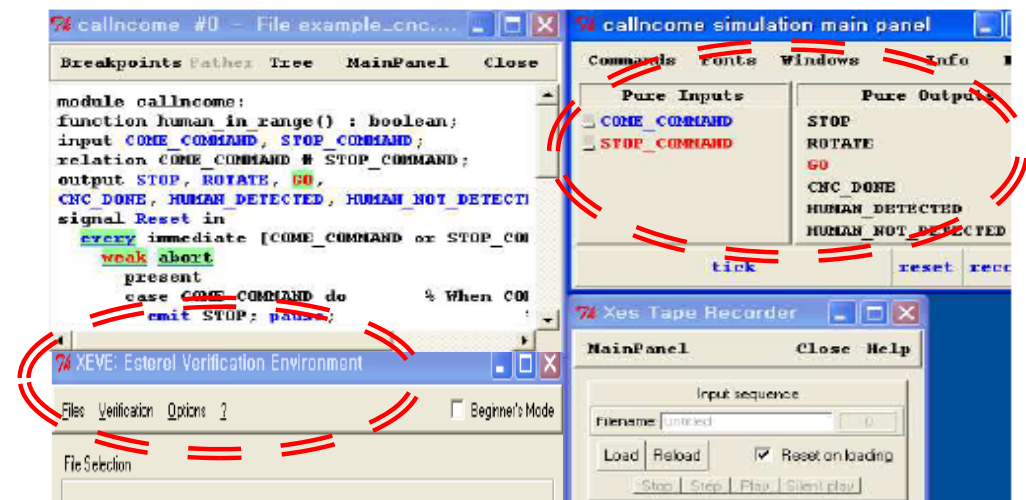
# Advantage of a Reactive PL

## Lessons Learned



- We uncovered subtle bugs which decrease the accuracy of detecting a user
- Implementing preemption in C++ is error prone.

- Esterel enables **clear interactions among the components**
- Esterel has **formal semantics** as Mealy machine, which allows rigorous analysis such as model checking





- After all, SALT decided **not** to adopt re-engineered robot sw in their robot prototype ☹
- Excuses are
  - Overhead of using a new language
    - Most robot developers are not from CS field
  - Inability to optimize final code manually
    - For consumer products, resource constraints are still major issues
  - Version discrepancy
    - While re-engineering was going on at POSTECH, SALT constantly add/updated features, which our re-engineered code did not cover

### ■ A Case Study of Re-engineering Home Service Robot

- Based on the three engineering principles, we designed a new SA and re-engineered existing source code.
- By this re-engineering, interactions among the components became visible and the responsibility of behaviors could be assigned to components clearly, which enhance the reliability
- By this re-engineering, we can apply model checking technique to improve the reliability of the control plane

### ■ Future work

- Resource management problem
- Guideline for reverse-engineering