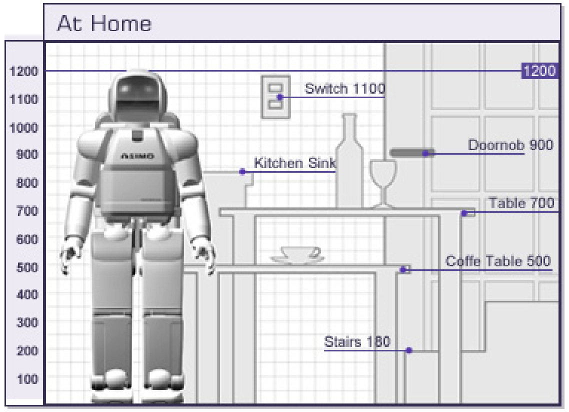
Re-engineering Home Service Robots Improving Software Reliability: A Case Study

Moonzoo Kim, etc

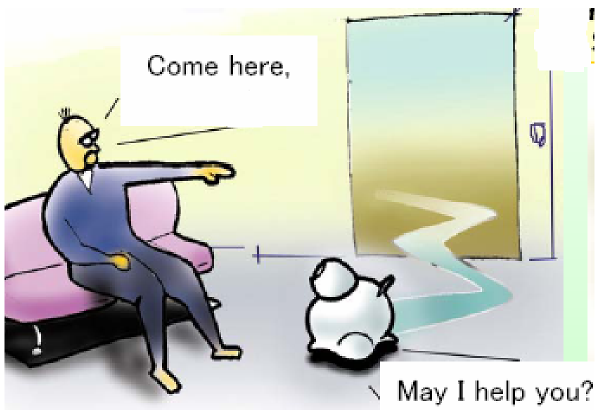
Agenda

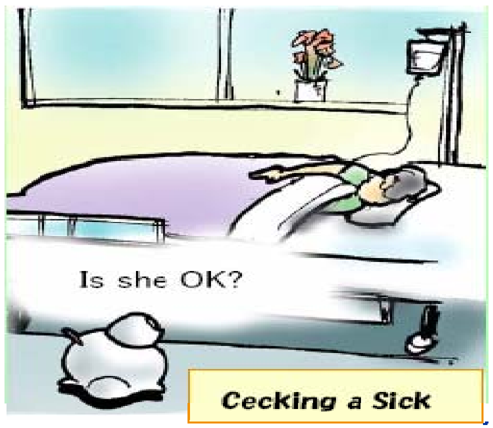
- Introduction
- Re-engineering Software Architecture
- Control Plane Re-engineering
- Data Plane Re-engineering
- Lessons Learned

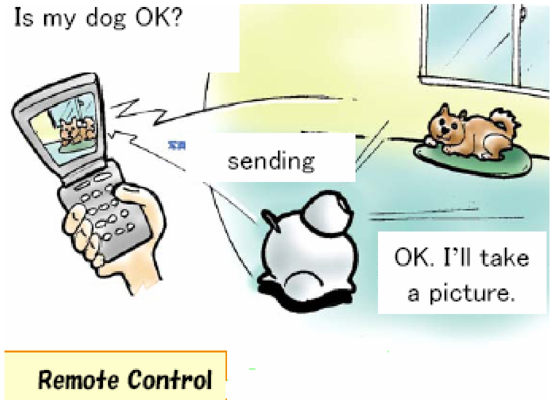
- Designed for providing various services to human user
  - Service areas : home security, patient caring, cleaning, etc
  - Markets for home service robots are still being formed



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



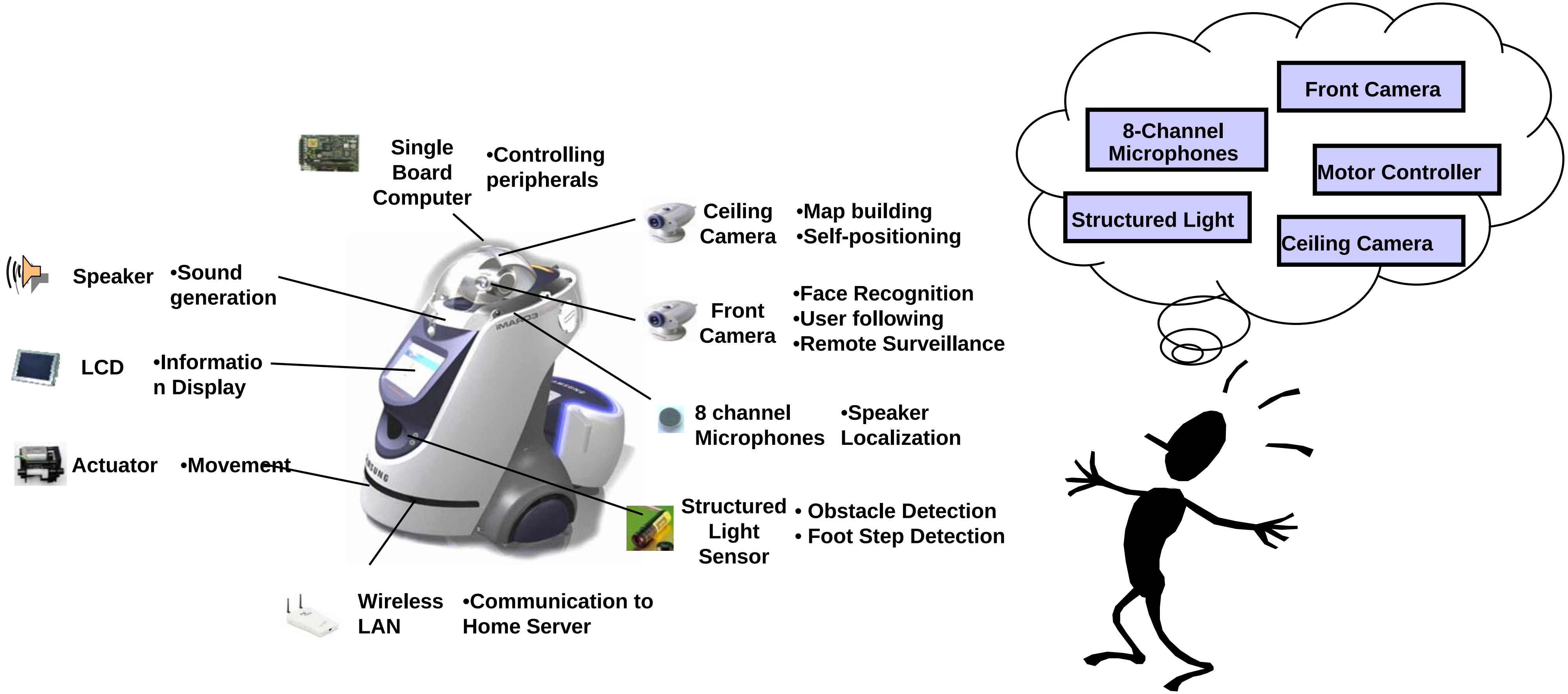




- SAIT started development of SHR00 from 2002
  - 4 separate teams (13 persons)
    - Vision recognition, speech recognition, simultaneous localization and mapping (SLAM), actuator
- Both SHR00 and SHR50 suffered feature interaction problems
  - SAIT decided to develop SHR100 from scratch
- SAIT requested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

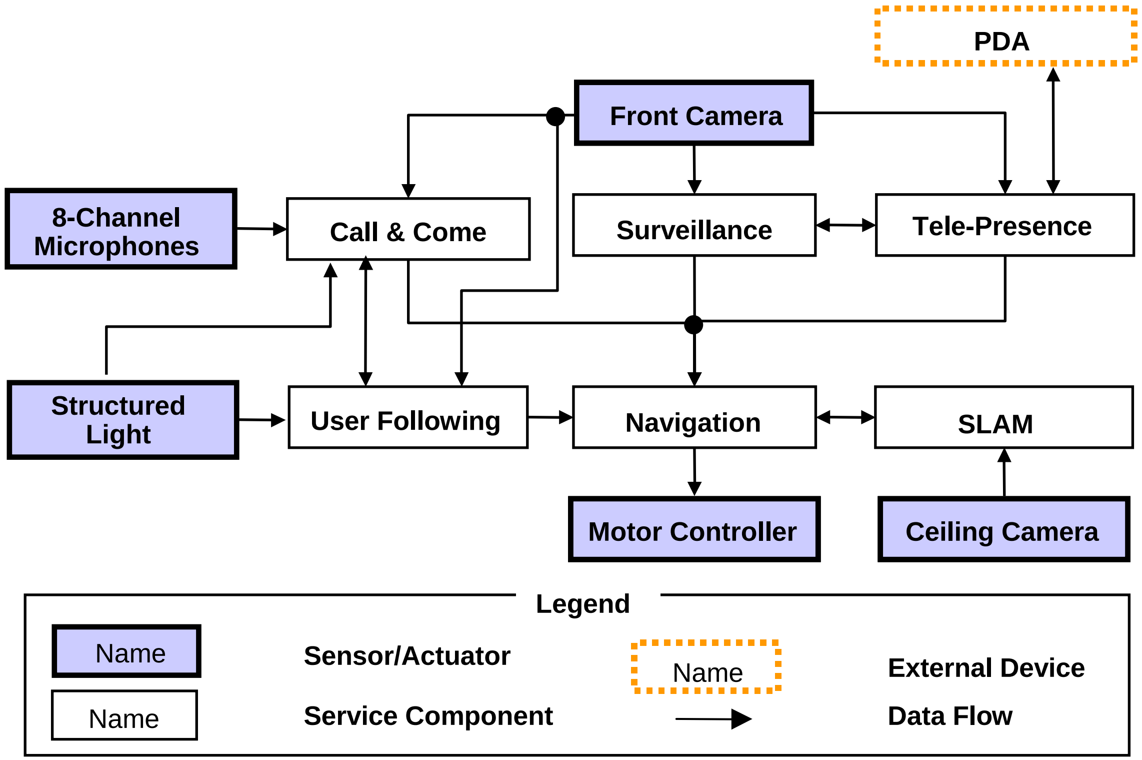
Components of Home Service Robots

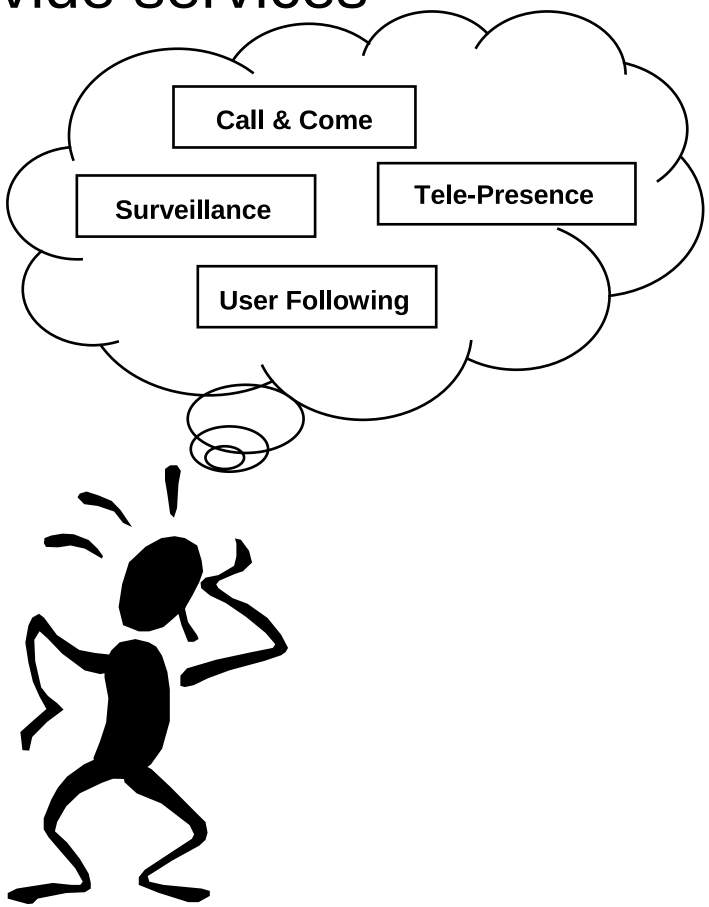
- Robots are created based on various **technical components**
  - Speech recognizer, vision recognizer, actuator, etc



Integration of Components

- 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

- 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

Proposed Approach

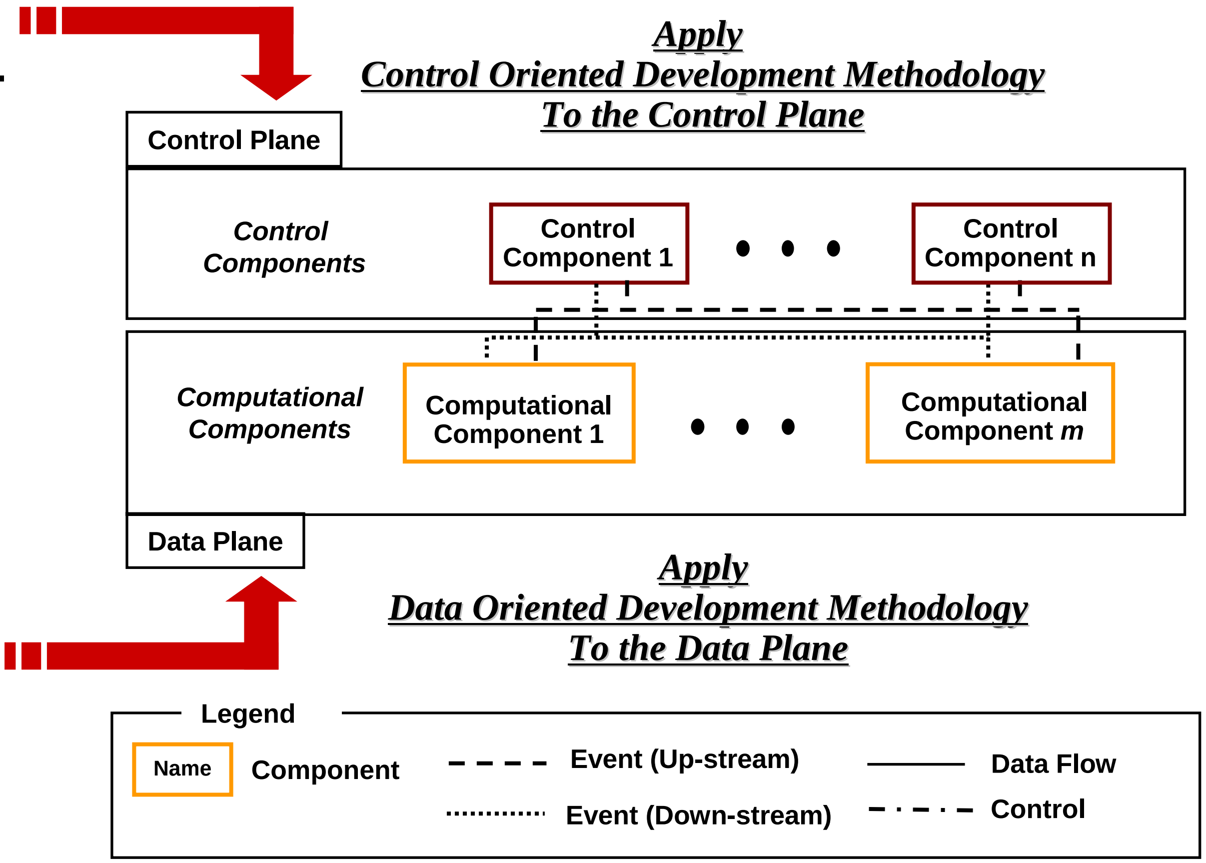
- Re-engineering based on the following three **principles**

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.

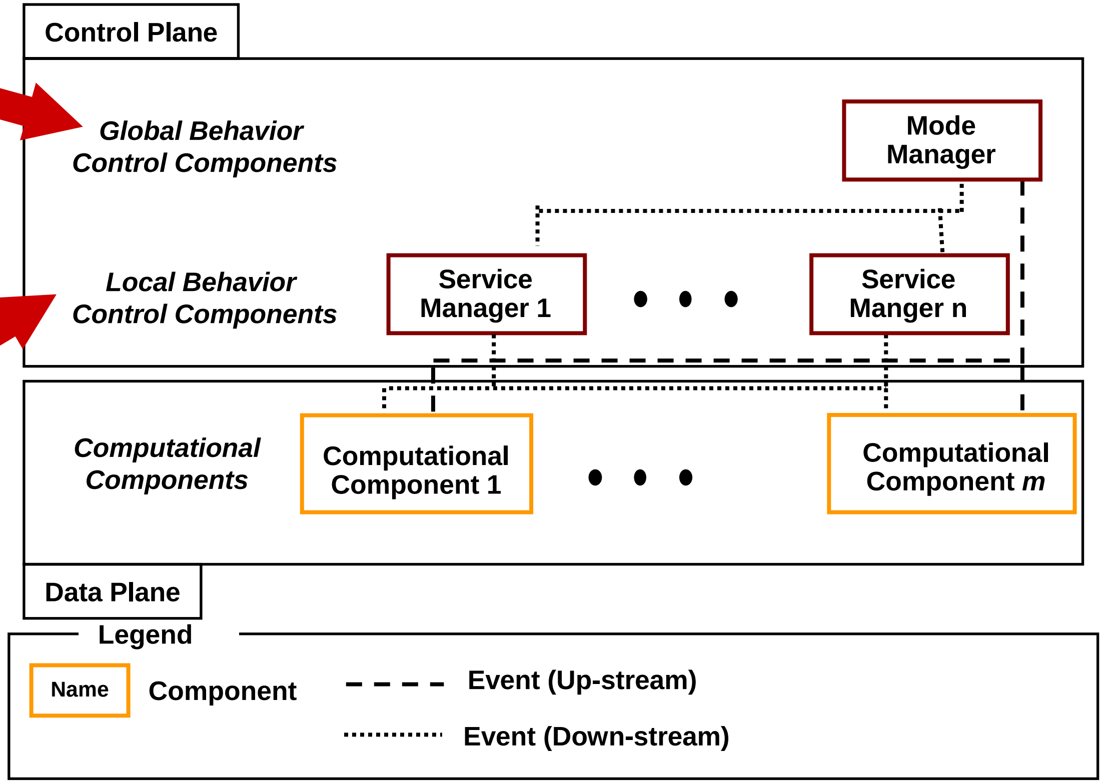
The second class of data is **computational data** for handling robot function.
: *efficient computation* is the most important goal.



Principle2: Separation of Local Behaviors from Global Behaviors

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

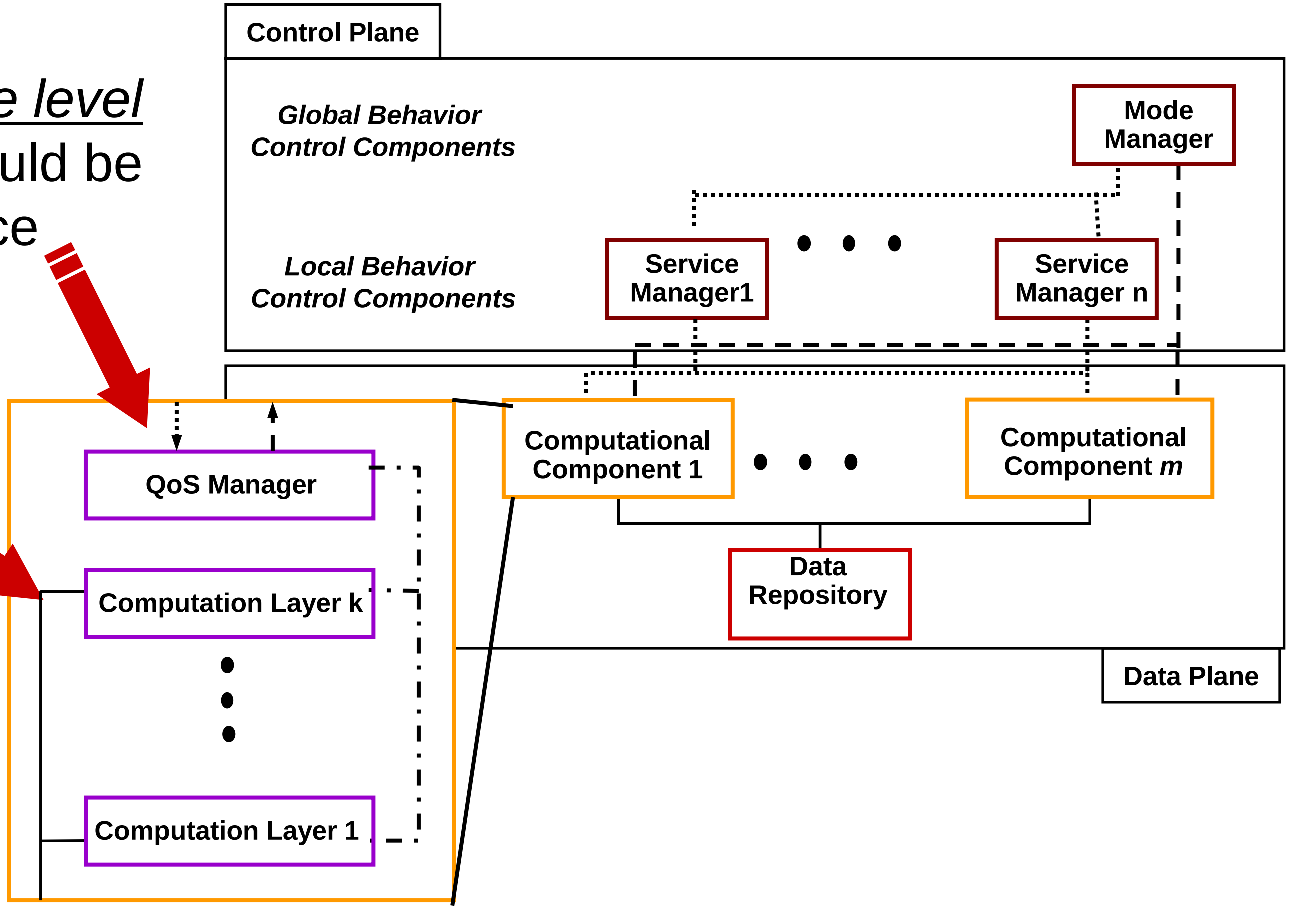
Service manager components defines the *behavior of service feature* by controlling the computational components.



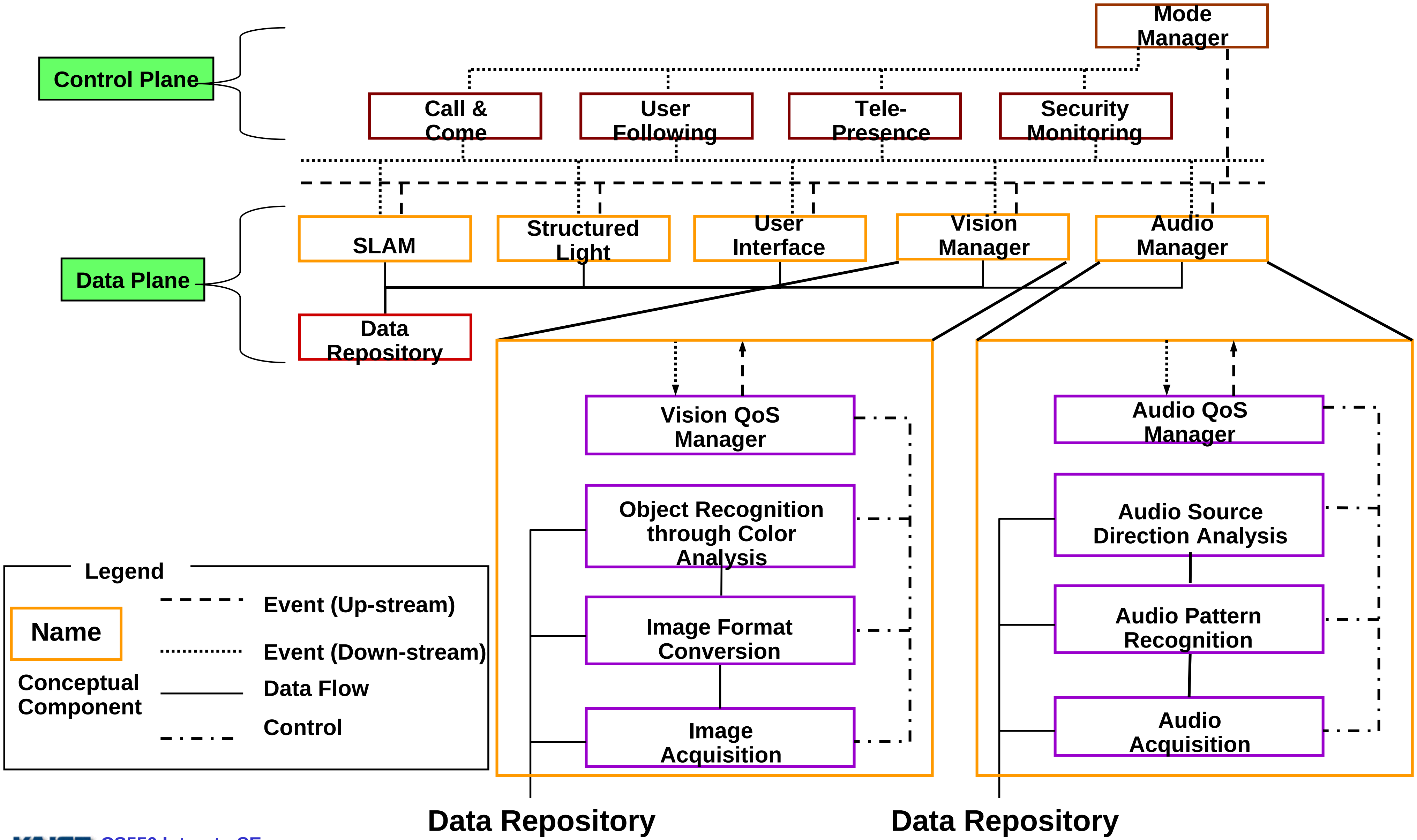
Principle3: Layering in Accordance with Data Refinement Hierarchy

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

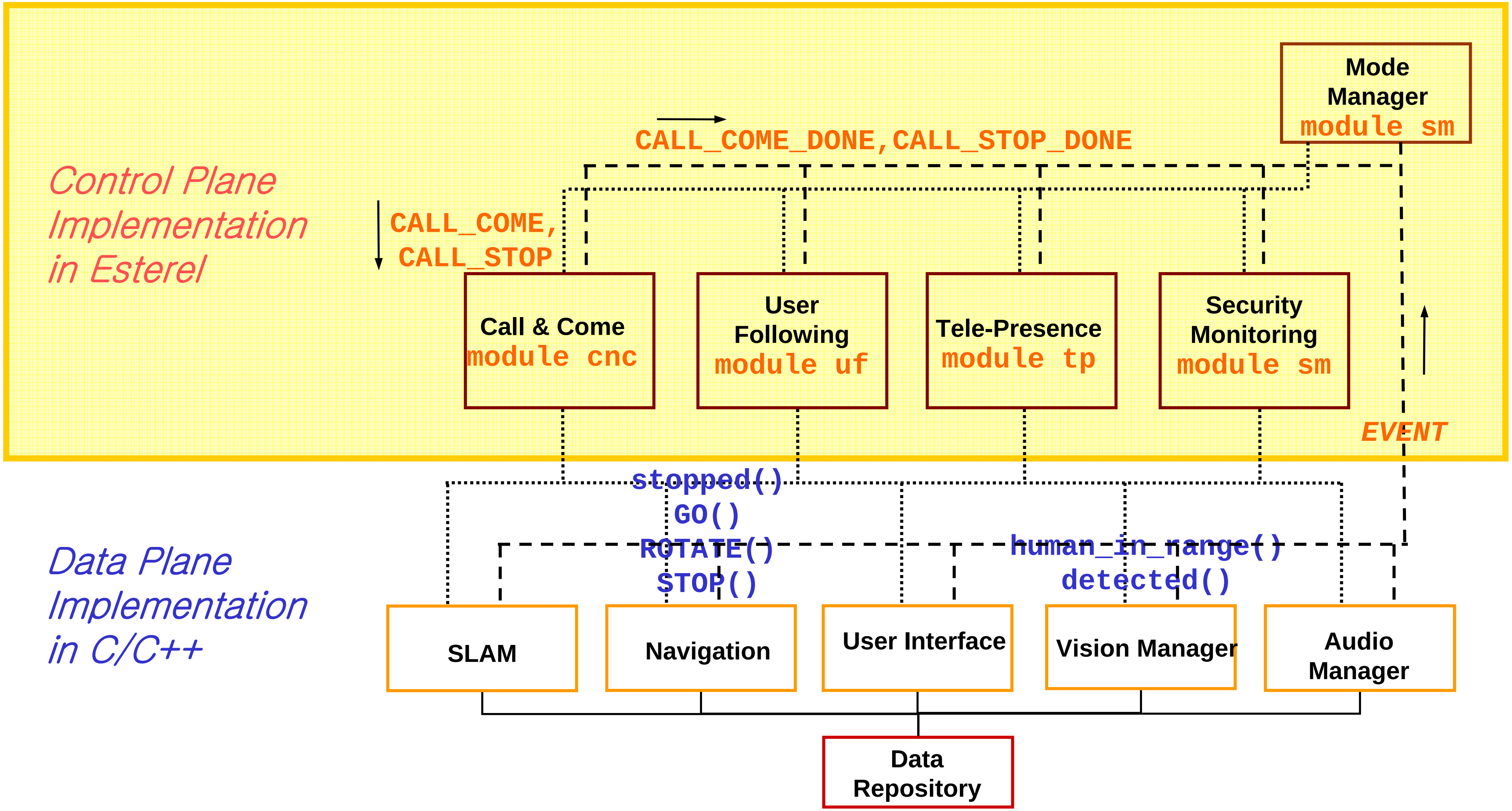
There exist **data refinement hierarchy** for data computation and different service features may use *different computational component layers*.



New Software Architecture *Re-engineering Software Architecture*



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



Re-engineering Control Plane (2/3)

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

New Commands (lines 10–18)

```
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_befO−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

```
module control_plane: % Control Plane
input EVENT: integer;
output STOP,ROT,GO,CC_DONE,CS_DONE,DET,N_DET;
signal CALL_COME, CALL_STOP in
run mode_man||run cnc||run uf||run tp||run sm;
end signal
end module

module cnc: % Call and Come service
function human_in_range() : boolean;
input CALL_COME,CALL_STOP; %come,stop commands
output STOP,ROT,GO,CC_DONE,CS_DONE,DET,N_DET;
var mv:=false:boolean,n:integer in
    every immediate [CALL_COME or CALL_STOP ] do
        present
        case CALL_COME do % come command
                mv := true;
                emit STOP; pause;
                run rot_det;
                ...
                emit CC_DONE;pause;
        case CALL_STOP do % stop command
                emit STOP;
                if mv=true then emit CS_DONE;
                else mv:=true;pause;run rot_det end if;
        end present;
        mv := false;
    end every
end var
end module
...
```

- Esterel handles a preemptive event e with a preemption operator

  EVERY e DO statements END EVERY.

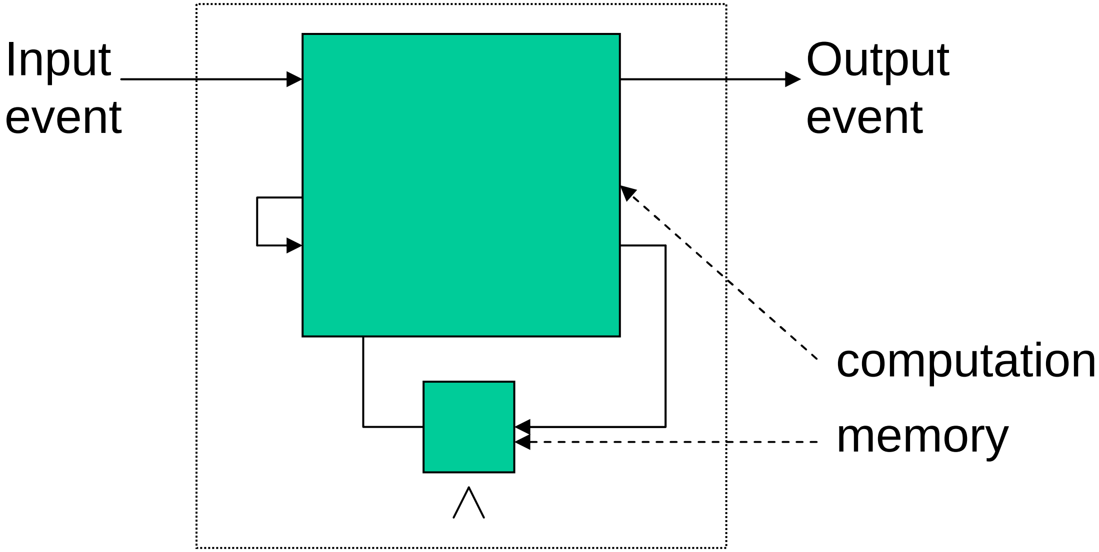
- Interactions among Esterel modules are clearly defined via events

  PRESENT CASE e DO statements END PRESENT

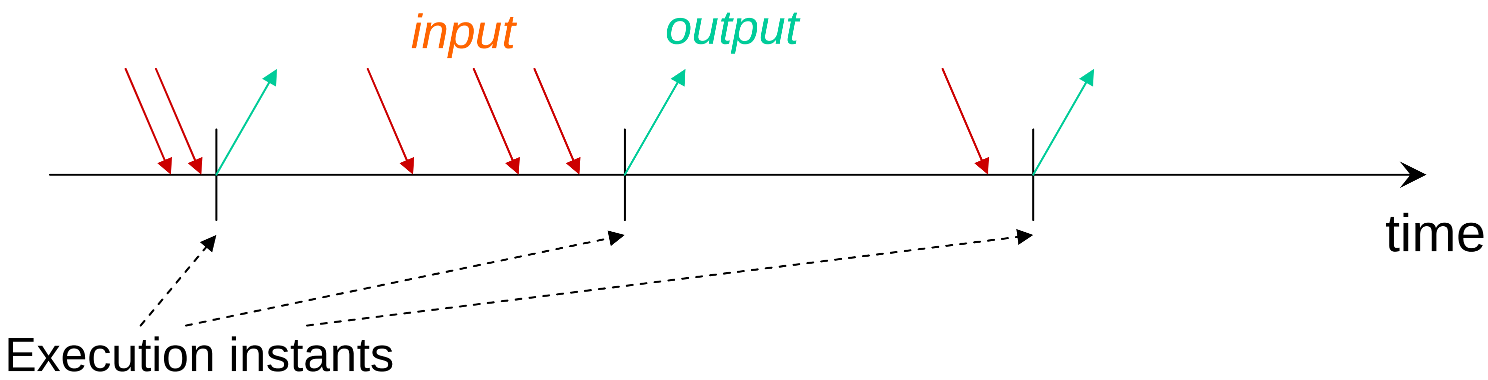
- Submodule can be conveniently utilized

  RUN module

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



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



Esterel Background (2/5)

- Synchronous language
- Structural imperative style
- Basic constructs
  - Classical control flow

    `p;q, p||q, loop p end`
  - Signals:

    `signal S in p end, emit S,`

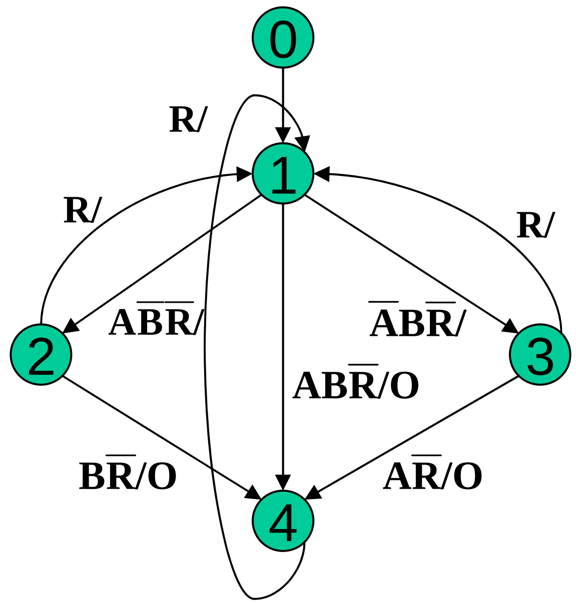
    `present S then p else q end`
  - Preemption

    `abort p when S, every s do p end every`
  - Exception handling

    `trap T in p end, exit T`

ABRO example

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

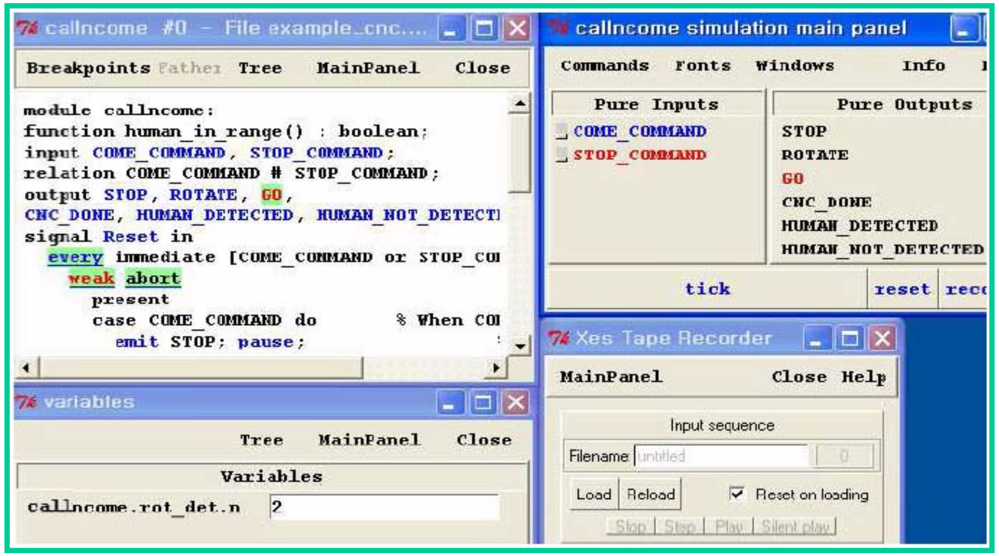


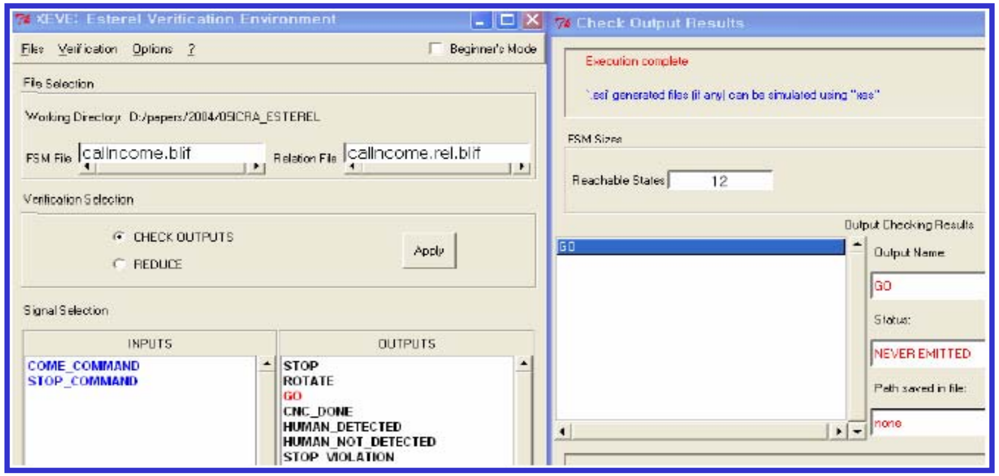

```
switch(state){
case 0: state=1; break;
case 1: if(!R)if(A)if(B) {O();state=4;}
                       else state=2;
                    else if(B)state=3;break;
case 2: if(R)state=1;
        else if(B){O();state=4;} break;
case 3: if(R)state=1;
        else if(A){O();state=4;} break;
case 4: if(R)state=1;break;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

Esterel Background (4/5)

Overview of Esterel Tools

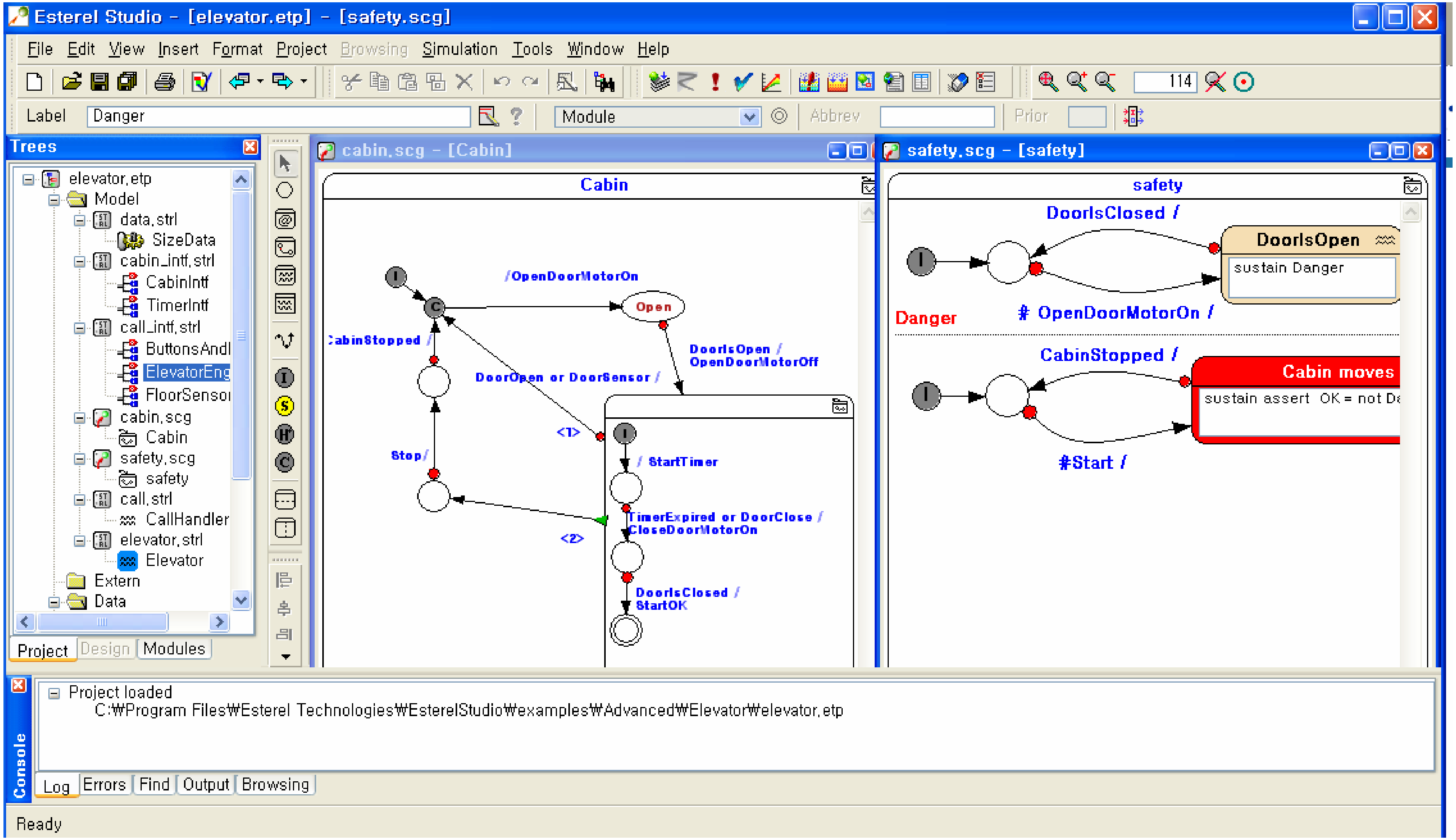
- The esterel **Compiler:**
  - C/VHDL/Verilog code generation.
  - interface between Esterel and C.
- The xes **Graphical Simulator:**
  - graphical interactive simulation
  - session recording/replay.
- The xeve **Model Checker:**
  - analyzes an Esterel program.
  - check presence of an output signal with given configuration of input signals.



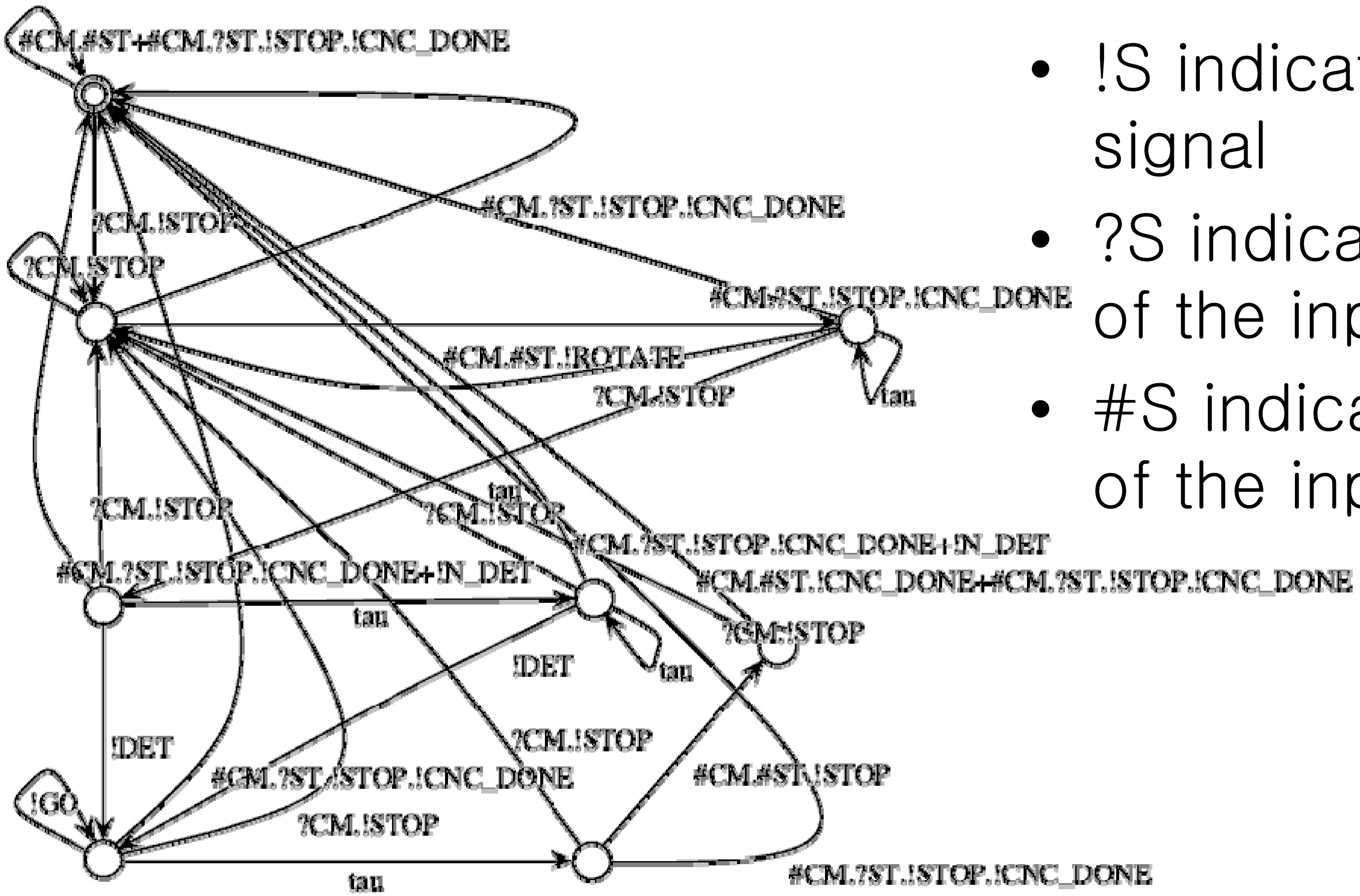


Esterel Background (5/5)

Commercial Esterel Studio 5.21



Behavior of CC

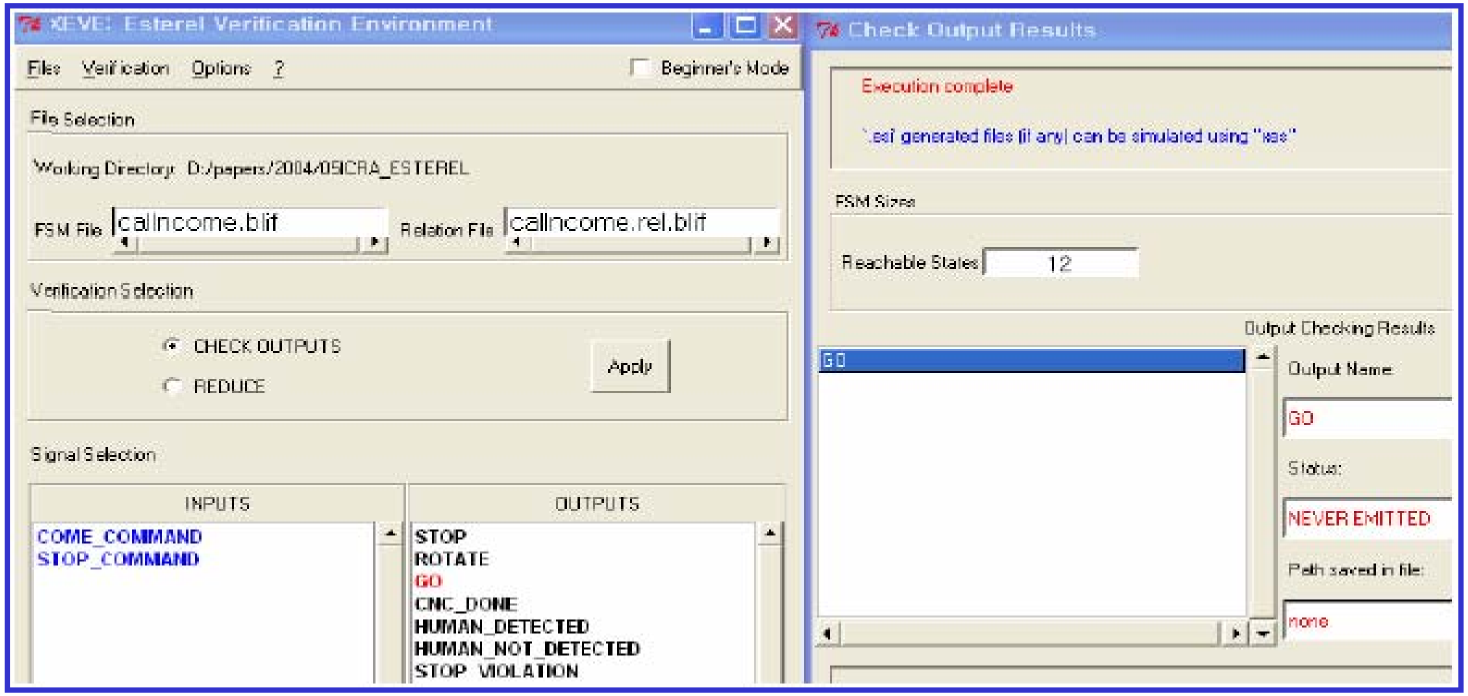


- !S indicates output signal
- ?S indicates presence of the input signal S
- #S indicates absence of the input signal S

Formal Verification of Stopping Behaviors (2/5)

- 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

- 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 had 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

```
module observer:
input STOP_COMMAND,COME_COMMAND,ROTATE,STOP,GO;
output STOP_VIOLATION;
weak abort
  every immediate STOP_COMMAND do
    present STOP then
      loop
        present [ROTATE or GO]
          then emit STOP_VIOLATION;
        end present;
        pause;
      end loop;
    end present
    emit STOP_VIOLATION;
  end every
when COME_COMMAND;
end module
```

Re-engineering Data Plane (1/2)

Layered Implementation of Vision Manager

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

Interface

```
class Layer3 {
protected :
   Layer2 *lowerLayer;

public :
  virtual bool L3Service()= 0;
  void setLowerLayer(Layer2 *l){
       lowerLayer = l; }
}
```

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

public :
  virtual bool L2Service()= 0;
  void setLowerLayer(Layer1 *l){
      lowerLayer = l; }
}
```

Implementation

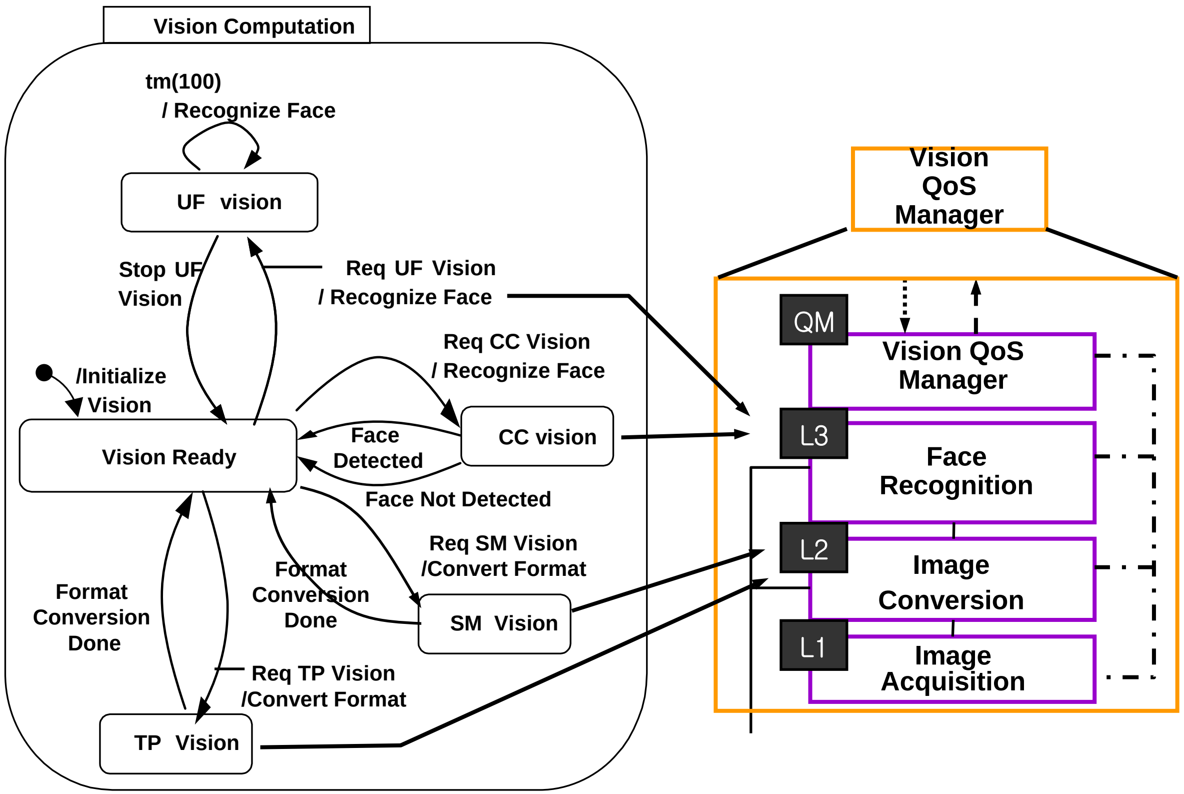
```
class Vision_L3_FaceRecognition
: public Layer3 {
public :
virtual bool L3Service()
{…
  if(lowerLayer->L2Service()){
    …
    if(m_faceRec.Rec()){
      DR::setData(m_facePattern);
… }
```

```
class Vision_L2_FormatConversion
: public Layer2 {
public :
virtual bool L2Service()
{…
  if(lowerLayer->L1Service()){
    …
    if(m_frmtConversion.Conv()){
      DR::setData(m_imgFormat);
… }
```

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.

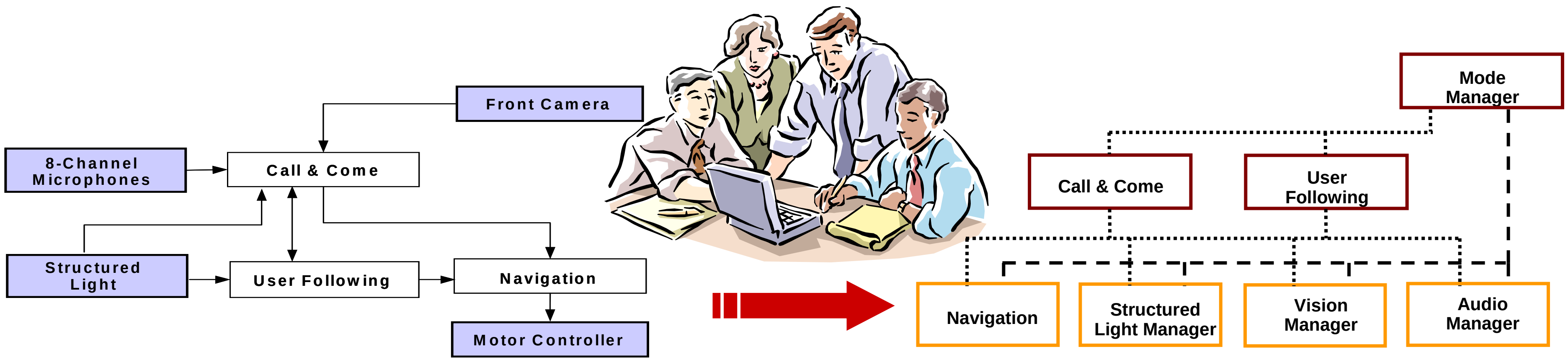


Necessity of Re-engineering

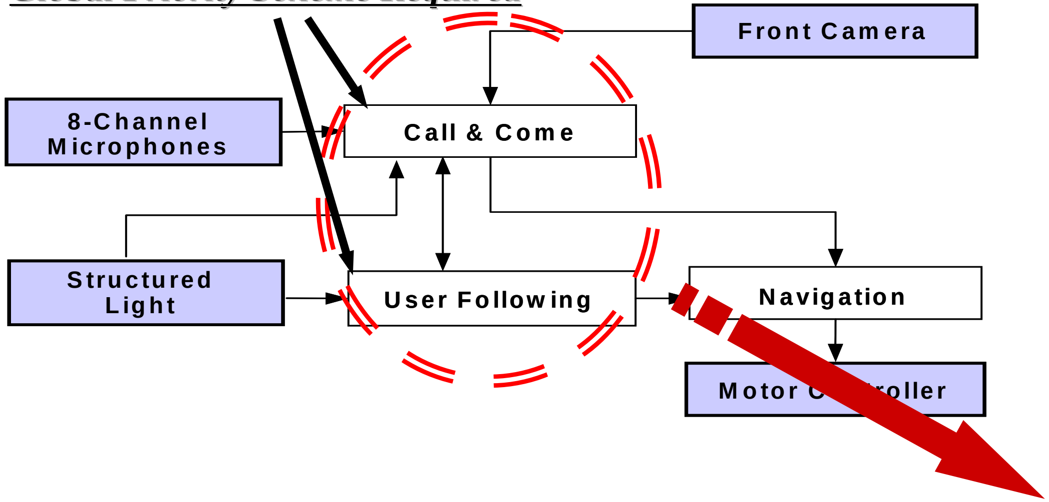
- From the experience of re-engineering SHR100, we are convinced that ***re-engineering is essential***

  - 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.

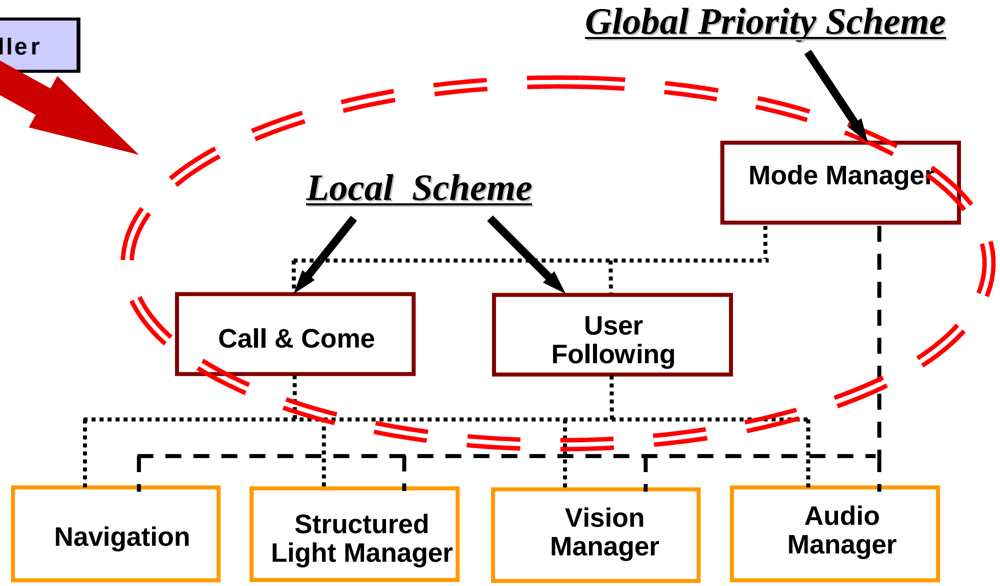


Separation of Priority Management



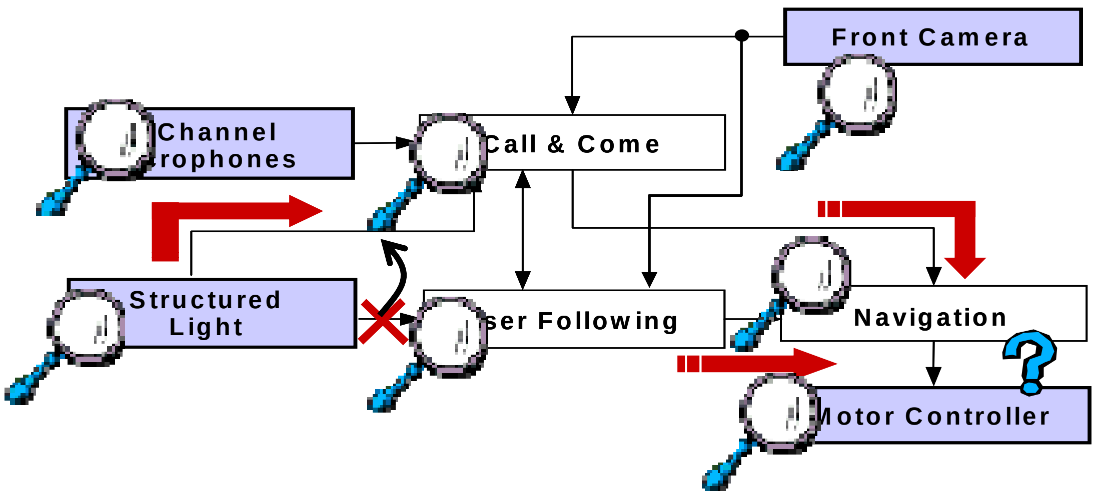
- We found that unclear global priority scheme was one of the primary causes of feature interaction problems.

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



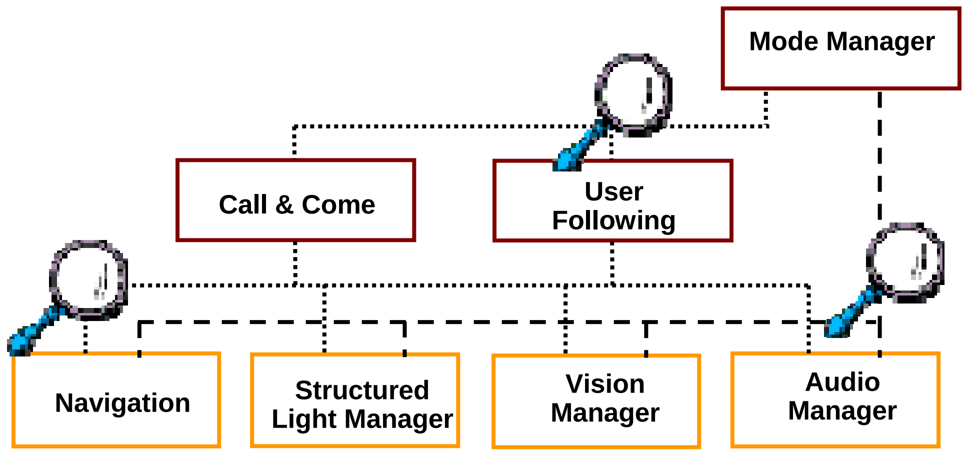
Needs of Monitoring Capability

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

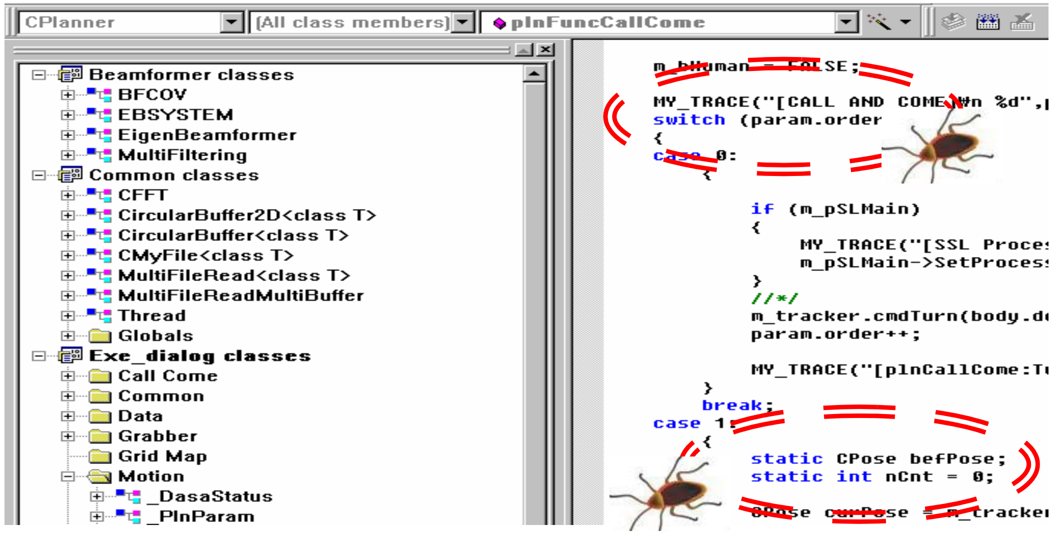


-Determining where to put probes is difficult, if the role of each component and the ways they interact each other are not clear

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



Advantage of a Reactive PL

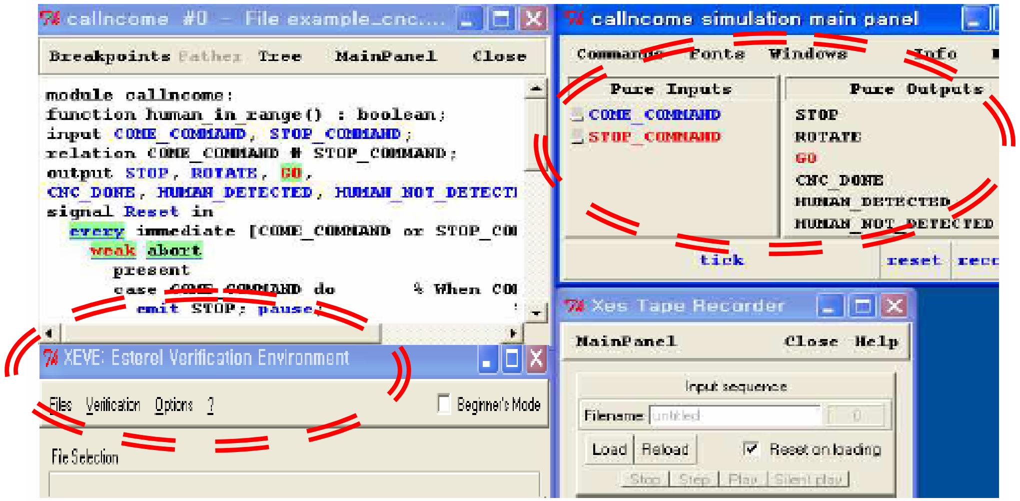


-We uncovered subtle bugs which decrease the accuracy of detecting a user

- Implementing preemption in C++ is error prone.

- Esterel enalbes ***clear interactions among the components***

- Esterel has ***formal semantics*** as Mealy machine, which allows rigorously analysis such as model checking



- After all, SAIT 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, SAIT constantly add/updated features, which our re-engineered code did not cover

Conclusion

- 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