PVSS & SMI++

Tools for the Automation of large distributed control systems

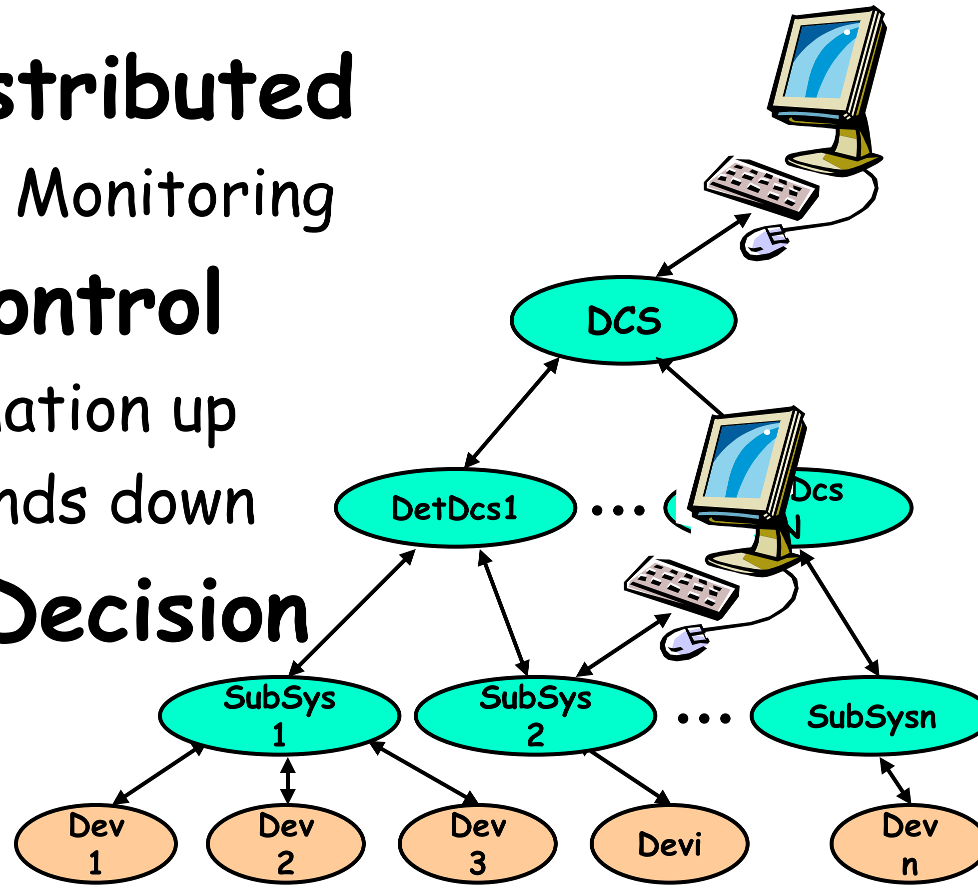
Clara Gaspar, PVSS Users Meeting, April 2005

Outline

- Some requirements of large control systems (for physics experiments)
- Control System Architecture
- Control Framework
  - SCADA: PVSS II
  - FSM toolkit: SMI++
- Some important features

Some Requirements…

- **Large number of devices/IOchannels**
- ➡ Need for:
  - **Parallel and Distributed**
    - Data acquisition & Monitoring
  - **Hierarchical Control**
    - Summarize information up
    - Distribute commands down
  - **Decentralized Decision Making**

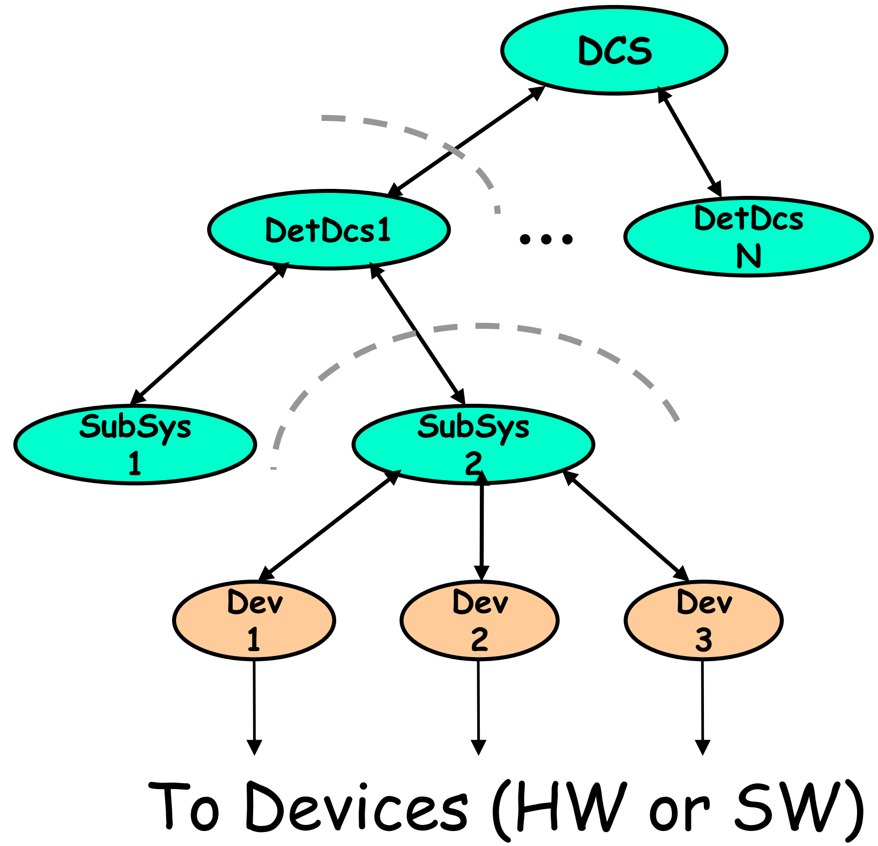


Some Requirements…

- **Large number of independent teams**
- **Very different operation modes**

➡ Need for:

- **Partitioning:** The capability of operating parts of the system independently and concurrently



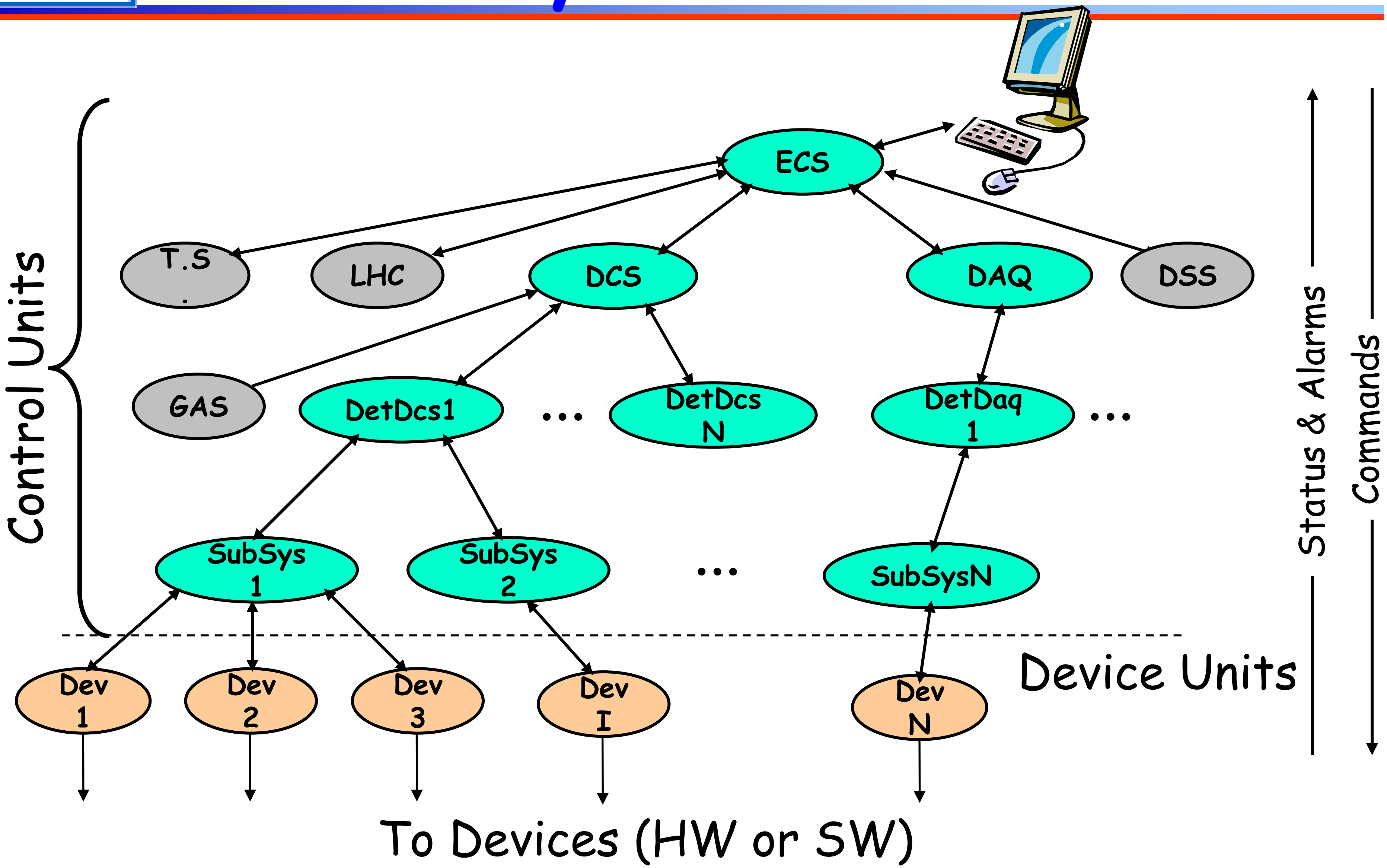
Some Requirements…

- **High Complexity**
- **Non-expert Operators**

➡ Need for:

  - **Full Automation** of:
    - Standard Procedures
    - Error Recovery Procedures
  - **Intuitive User Interfaces**

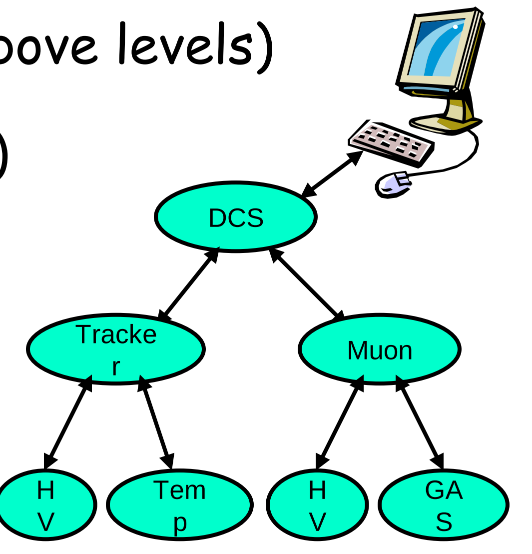
Control System Architecture



Clara Gaspar, April 2005

Control Units

- **Each node is able to:**
  - Summarize information (for the above levels)
  - "Expand" actions (to the lower levels)
  - Implement specific behaviour & Take local decisions
    - Sequence & Automate operations
    - Recover errors
  - Include/Exclude children (i.e. partitioning)
    - Excluded nodes can run is stand-alone
  - User Interfacing
    - Present information and receive commands



The Control Framework

- **The JCOP Framework* is based on:**
  - SCADA System - PVSSII for:
    - Device Description (Run-time Database)
    - Device Access (OPC, Profibus, drivers)
    - Alarm Handling (Generation, Filtering, Masking, etc)
    - Archiving, Logging, Scripting, Trending
    - User Interface Builder
    - Alarm Display, Access Control, etc.
  - SMI++ providing:
    - Abstract behavior modeling (Finite State Machines)
    - Automation & Error Recovery (Rule based system)

Device Units

Control Units

*Please See Talk S11-1

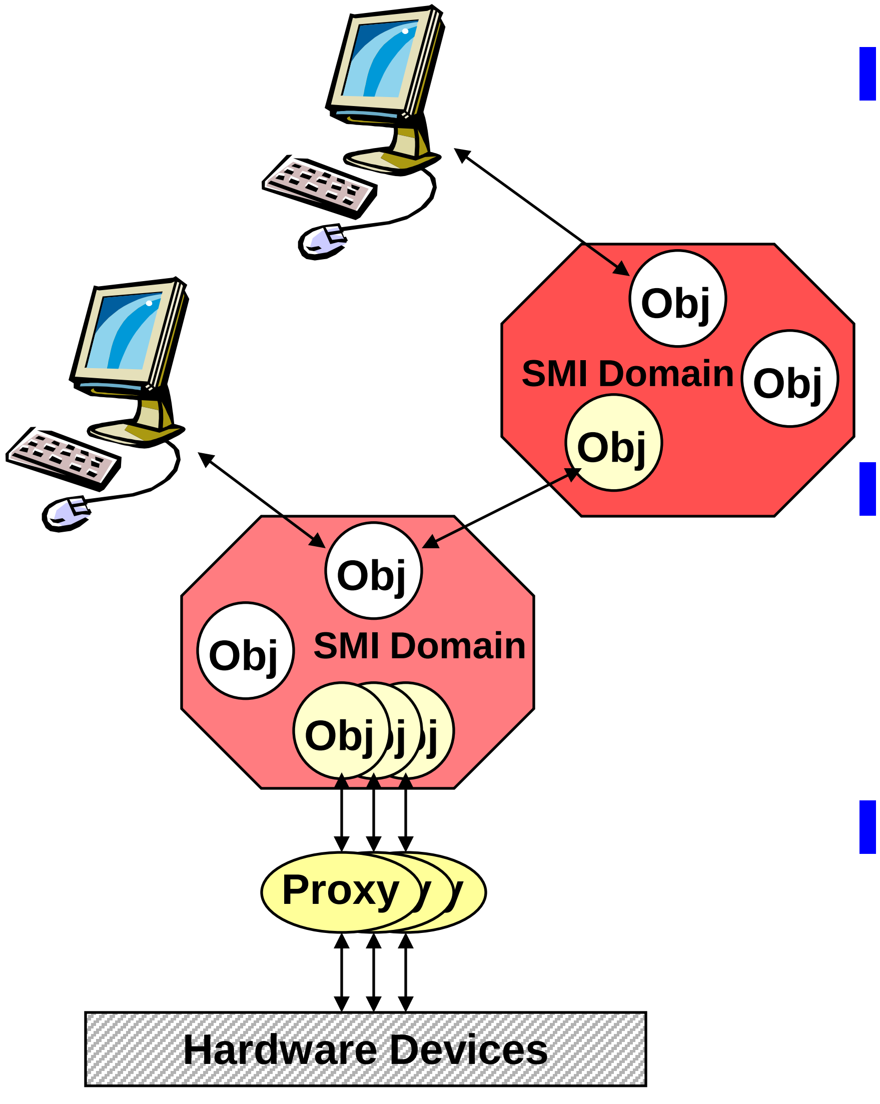
SMI++

- **Method**
  - Classes and Objects
    - Allow the decomposition of a complex system into smaller manageable entities
  - Finite State Machines
    - Allow the modeling of the behavior of each entity and of the interaction between entities in terms of STATES and ACTIONS
  - Rule-based reasoning
    - Allow Automation and Error Recovery

SMI++

- **Method (Cont.)**
  - SMI++ Objects can be:
    - Abstract (e.g. a Run or the DCS)
    - Concrete (e.g. a power supply or a temp. sensor)
  - Concrete objects are implemented externally either in "C", in C++, or in PVSS (ctrl scripts)
  - Logically related objects can be grouped inside "SMI domains" representing a given sub-system

SMI++ Run-time Environment



- Device Level: Proxies
  - C, C++, PVSS ctrl scripts
  - drive the hardware:
    - deduceState
    - handleCommands
- Abstract Levels: Domains
  - Internal objects
  - Implement the logical model
  - Dedicated language
- User Interfaces
  - For User Interaction

SMI++

- **SMI++ - The Language**
  - SML -State Management Language
    - Finite State Logic
      - Objects are described as FSMs
        their main attribute is a STATE
    - Parallelism
      - Actions can be sent in parallel to several objects.
        Tests on the state of objects can block if the objects are still "transiting"
    - Asynchronous Rules
      - Actions can be triggered by logical conditions on the state of other objects

SML example

- **Device:**

```
class: PowerSupply /associated
  state: UNKNOWN /dead_state
  state: OFF
    action : SWITCH_ON
  state: ON
    action : SWITCH_OFF
  state: TRIP
    action : RESET
    ...

object: PS1 is_of_class PowerSupply
```

- **Sub System:**

```
class: HighVoltage
  state: NOT_READY /initial_state
    action: GOTO_READY
      do SWITCH_ON PS1
      if ( PS1 in_state ON ) then
        move_to READY
      endif
      move_to ERROR
  state: READY
    when ( PS1 in_state TRIP ) do RECOVER
    action: RECOVER
      do RESET PS1
      do SWITCH_ON PS1
      ...
    action: GOTO_NOT_READY
    ...
  state: ERROR
    ...

object: SubDetHV is_of_class HighVoltage
```

SML example (many objs)

- Devices:

```
class: PowerSupply /associated
   state: UNKNOWN /dead_state
   state: OFF
      action : SWITCH_ON
   state: ON
      action : SWITCH_OFF
   state: TRIP
      action : RESET
      ...

object: PS1 is_of_class PowerSupply
object: PS2 is_of_class PowerSupply
object: PS3 is_of_class PowerSupply
...

objectset: PSS {PS1, PS2, PS3, ...}
```

- Objects can be dynamically included/excluded in a Set

- Sub System:

```
class: HighVoltage
  state: NOT_READY /initial_state
    action: GOTO_READY
      do SWITCH_ON all_in PSS
      if (all_in PSS in_state ON) then
        move_to READY
      endif
      move_to ERROR
  state: READY
    when ( any_in PSS in_state TRIP ) do RECOVER
    action: RECOVER
      do RESET all_in PSS
      do SWITCH_ON all_in PSS
      ...
    action: GOTO_NOT_READY
    ...
  state: ERROR
    ...

object: SubDetHV is_of_class HighVoltage
```

SML example (automation)

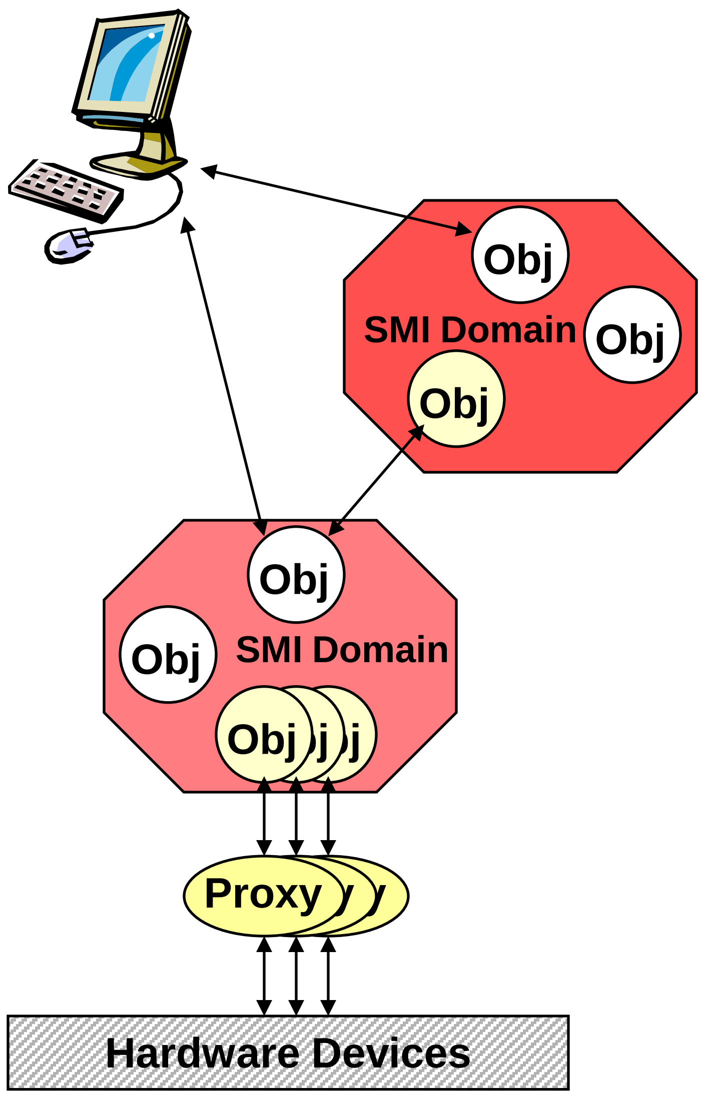
- External Device:

```
object: LHC::STATE /associated
  state: UNKNOWN /dead_state
  state: PHYSICS
  state: SETUP
  state: OFF
    ...
```

- Sub System:

```
object: RUN_CONTROL
  state: TEST_MODE
    when (LHC::STATE in_state PHYSICS) do PHYSICS
    action: PHYSICS
      do GOTO_READY SubDetHV
      ...
      move_to PHYSICS_MODE
  state: PHYSICS_MODE
    ...
```

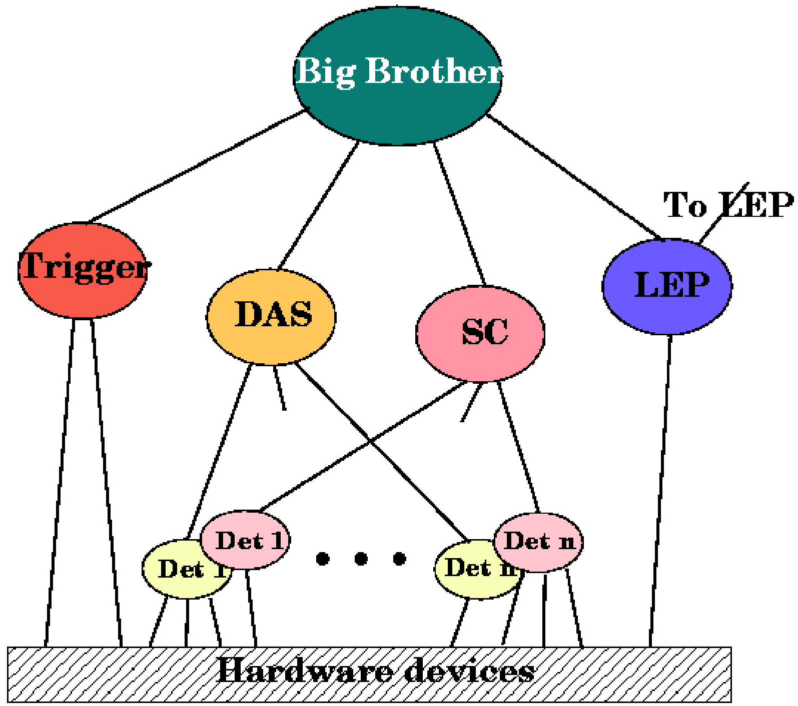
SMI++ Run-time Tools



- Device Level: Proxies
  - C, C++, PVSS ctrl scripts
  - Use a Run Time Library: **smirtl** To Communicate with their domain
- Abstract Levels: Domains
  - A C++ engine: **smiSM** - reads the translated SML code and instantiates the objects
- User Interfaces
  - Use a Run Time Library: **smiuirtl** To communicate with the domains
- All Tools available on:
  - Windows, Unix (Linux)
- All Communications are dynamically (re)established

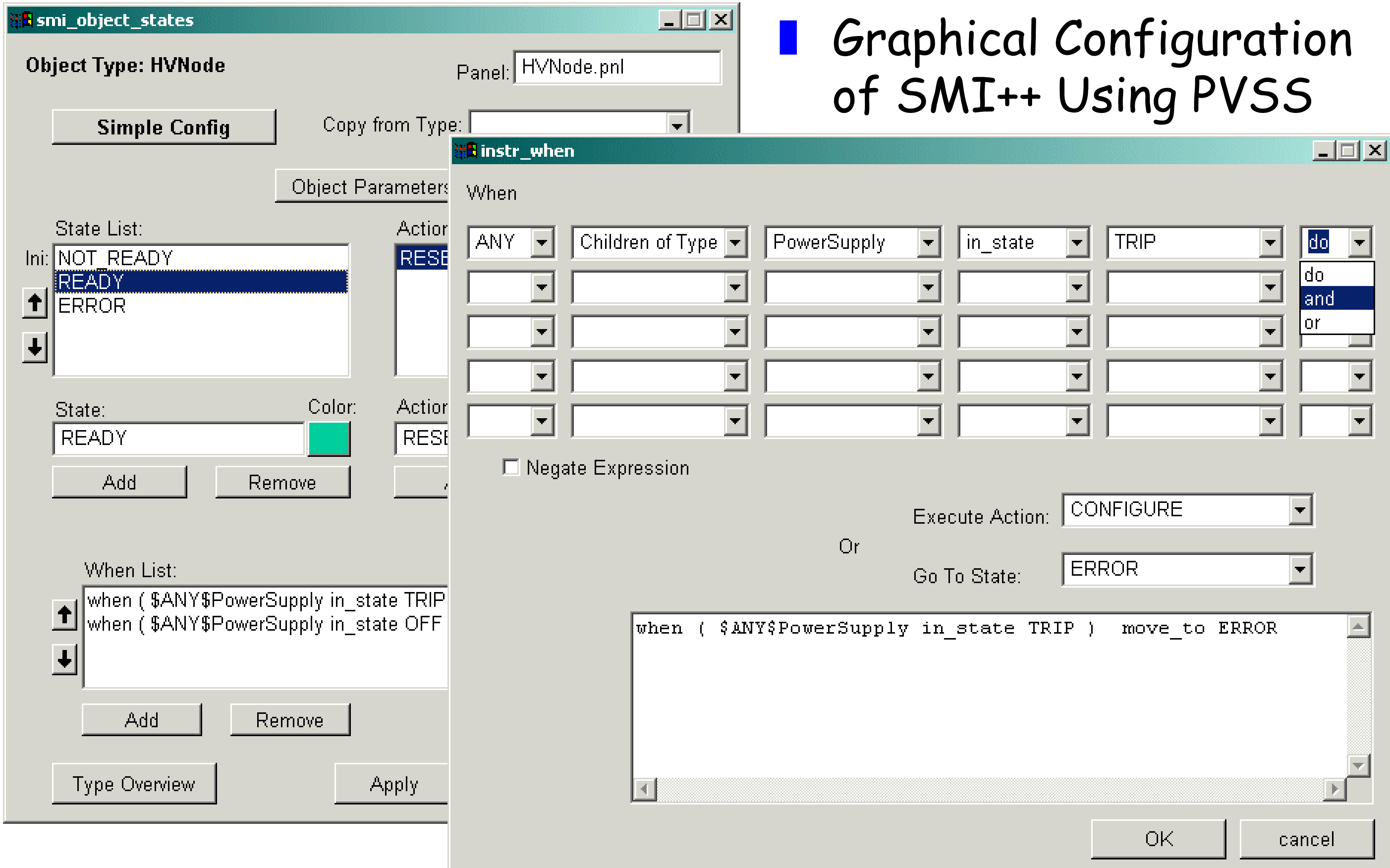
SMI++ History

- 1989: First implemented for DELPHI in ADA
  - DELPHI used it throughout the experiment
  - A Top level domain: Big-Brother automatically piloted the experiment
- 1997: Rewritten in C++
- 1999: Is used by BaBar for the Run-Control and high level automation
- 2002: Integration in PVSS for use at LHC

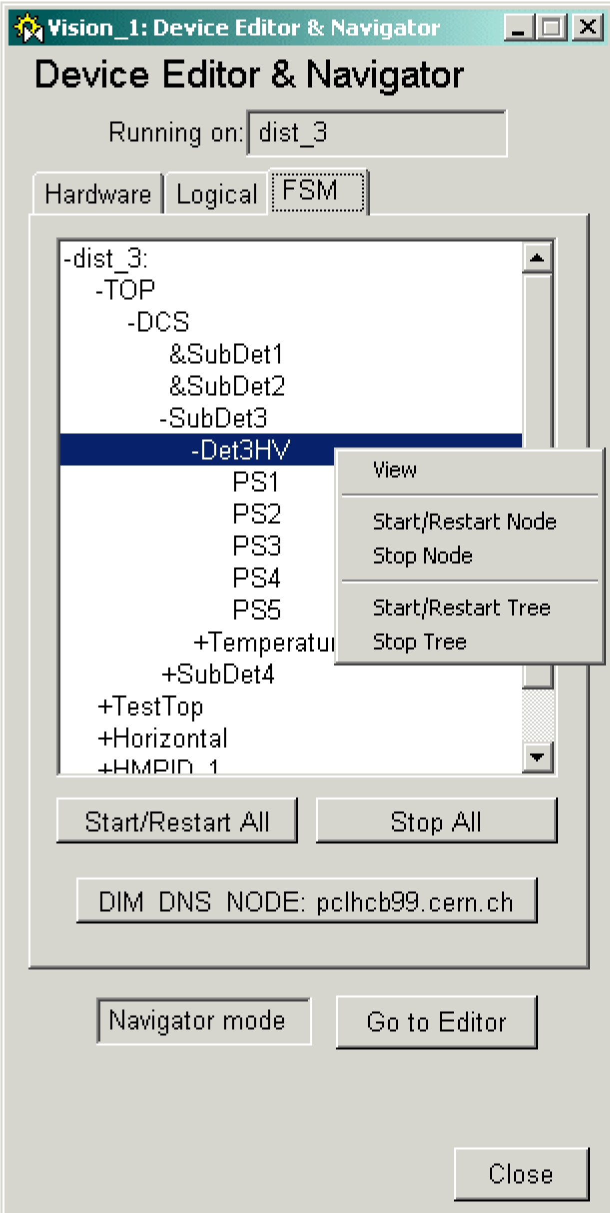


PVSS/SMI++ Integration

- Graphical Configuration of SMI++ Using PVSS



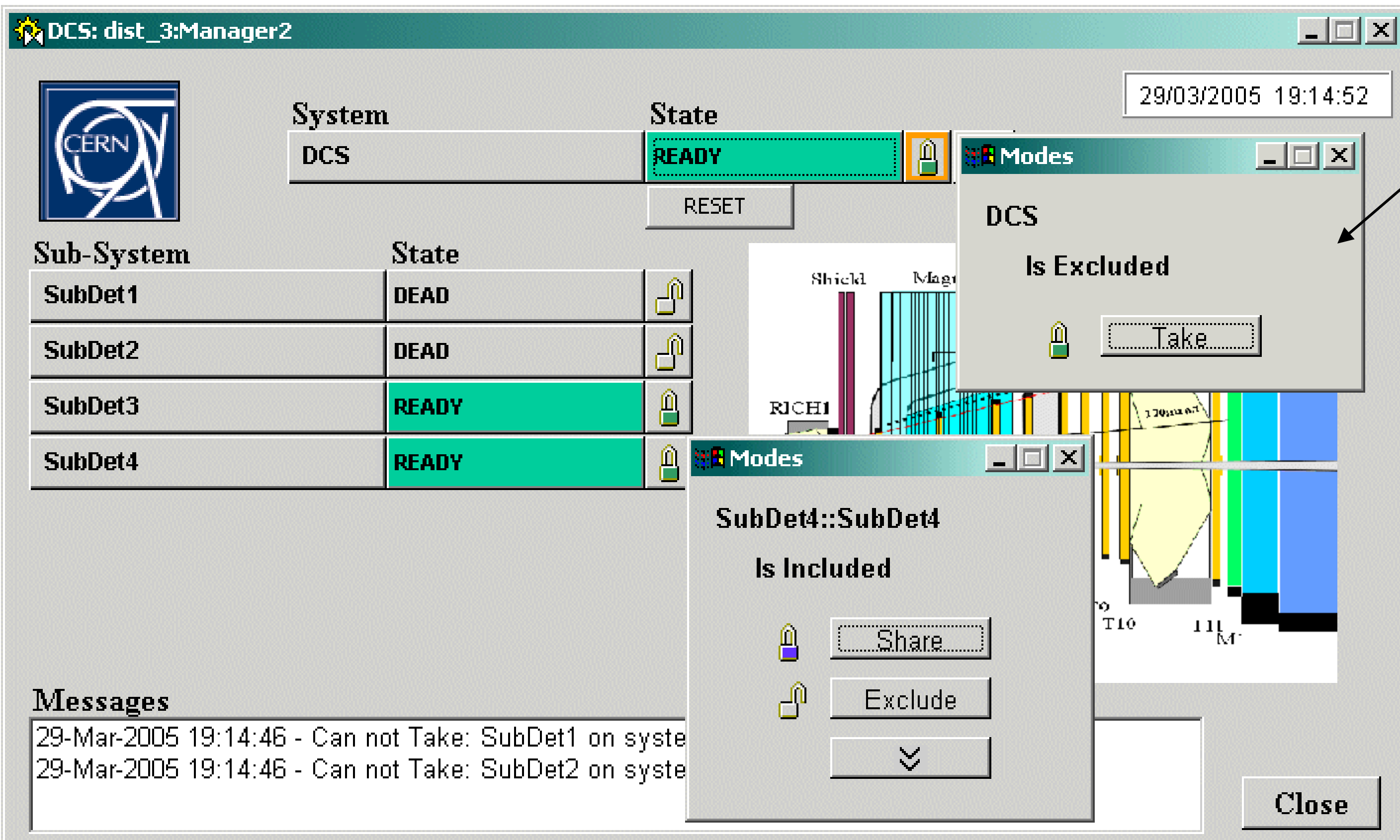
Building Hierarchies



- **Hierarchy of CUs**
  - Distributed over several machines
    - "&" means reference to a CU in another system
  - Editor Mode:
    - Add / Remove / Change Settings
  - Navigator Mode
    - Start / Stop / View

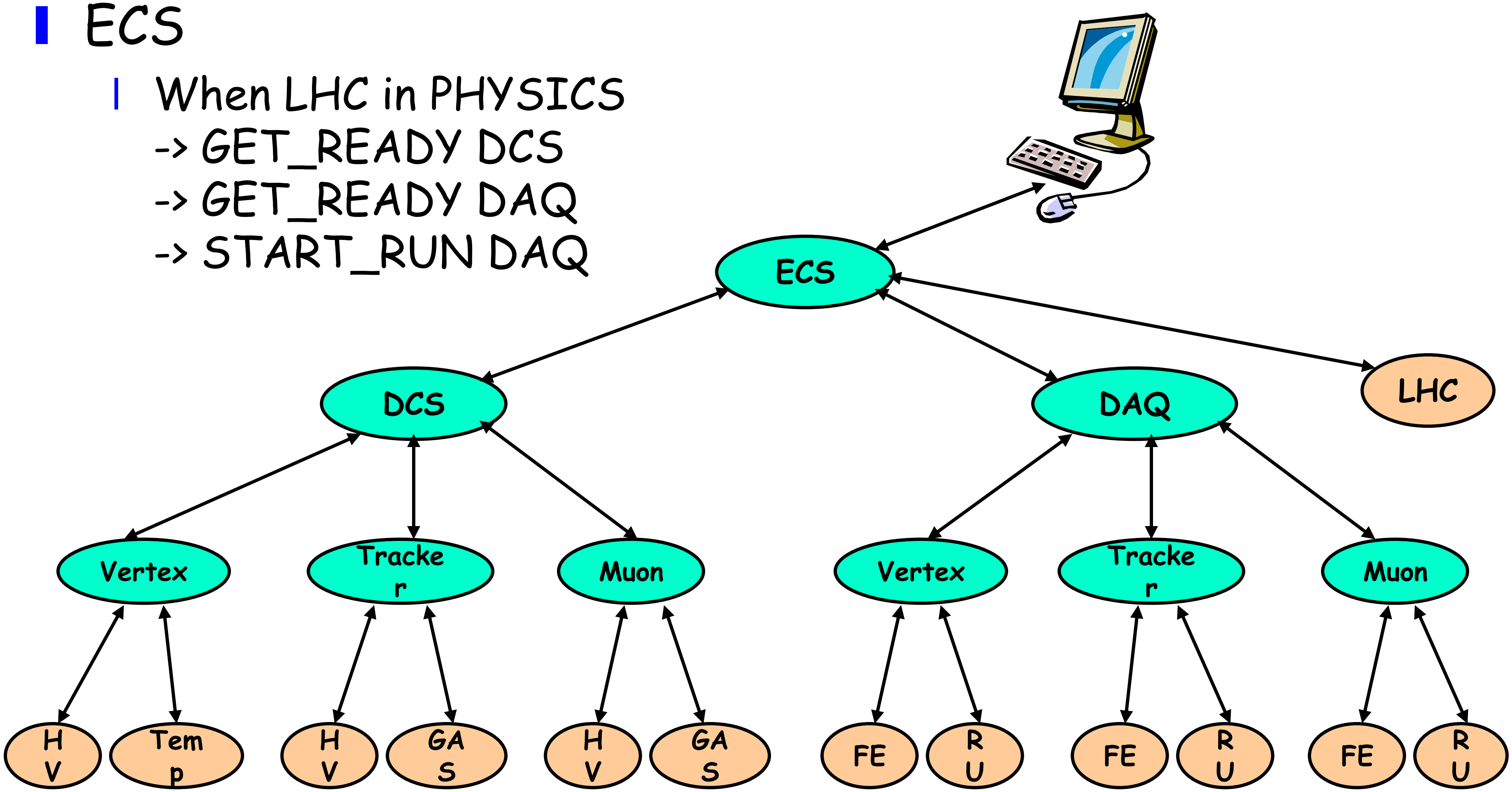
Control Unit Run-Time

- **Dynamically generated operation panels** (Uniform look and feel)
- Configurable User Panels



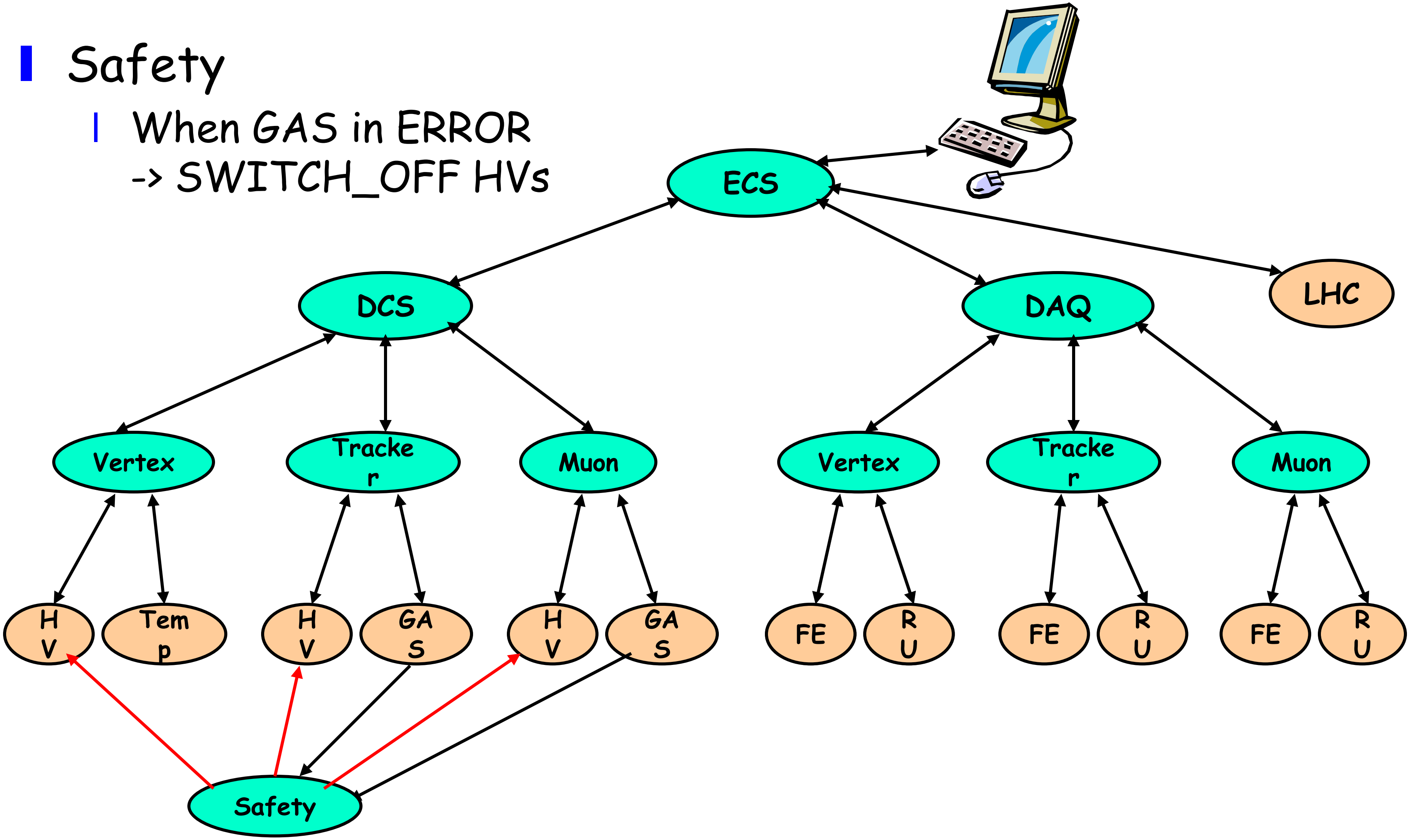
Full Experiment Control

- ECS
  - When LHC in PHYSICS
    -> GET_READY DCS
    -> GET_READY DAQ
    -> START_RUN DAQ



Parallel Hierarchies

- Safety
  - When GAS in ERROR -> SWITCH_OFF HVs



Features of PVSS/SMI++

- **Task Separation:**
  - SMI Proxies/PVSS Scripts execute only basic actions – No intelligence
  - SMI Objects implement the logic behaviour
  - Advantages:
    - Change the HW
      -> change only PVSS
    - Change logic behaviour
      sequencing and dependency of actions, etc
      -> change only SMI rules

Features of PVSS/SMI++

- **Error Recovery Mechanism**
  - Bottom Up
    - SMI Objects react to changes of their children
      - In an event-driven, asynchronous, fashion
  - Distributed
    - Each Sub-System recovers its errors
      - Each team knows how to recover local errors
  - Hierarchical/Parallel recovery
  - Can provide complete automation even for very large systems