SMI++

The Finite State Machine toolkit of the JCOP Framework

Clara Gaspar, February 2004

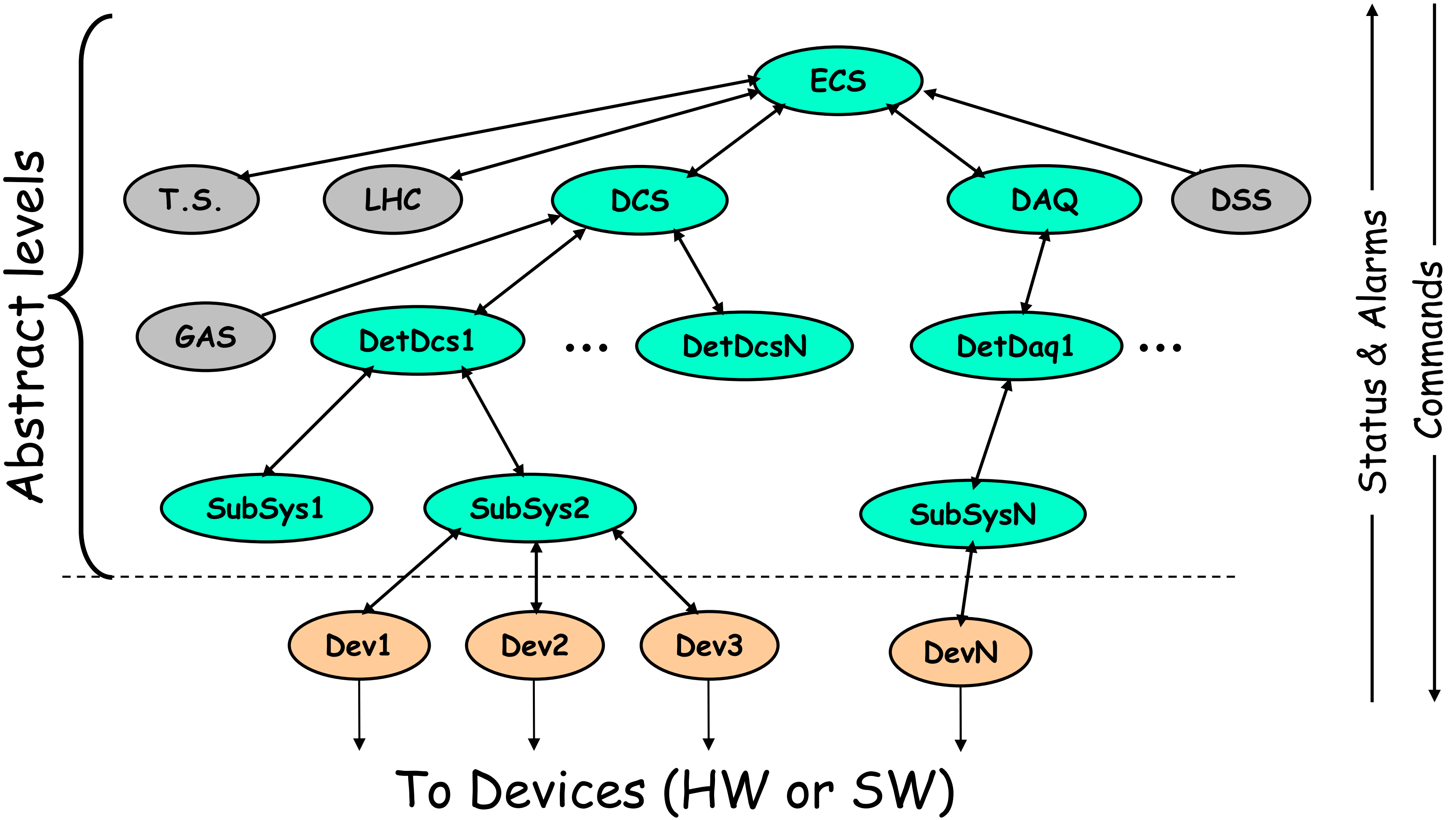
Outline

- **SMI++**
  - What is SMI++
  - Methodology
  - Tools
- **Framework <-> SMI++ Integration**
  - Device Units
  - Control Units
- **PVSS <-> SMI++ Integration**
  - Technical implementation

SMI++ History

- First implemented for DELPHI
  - by CERN DD/OC group
  - in ADA
- DELPHI used it for the control and automation of the complete experiment
  - SMI++ was then rewritten in C++
    - by B. Franek (and C. Gaspar)
- Being used by BaBar for the Run-Control and high level automation

Control System Architecture



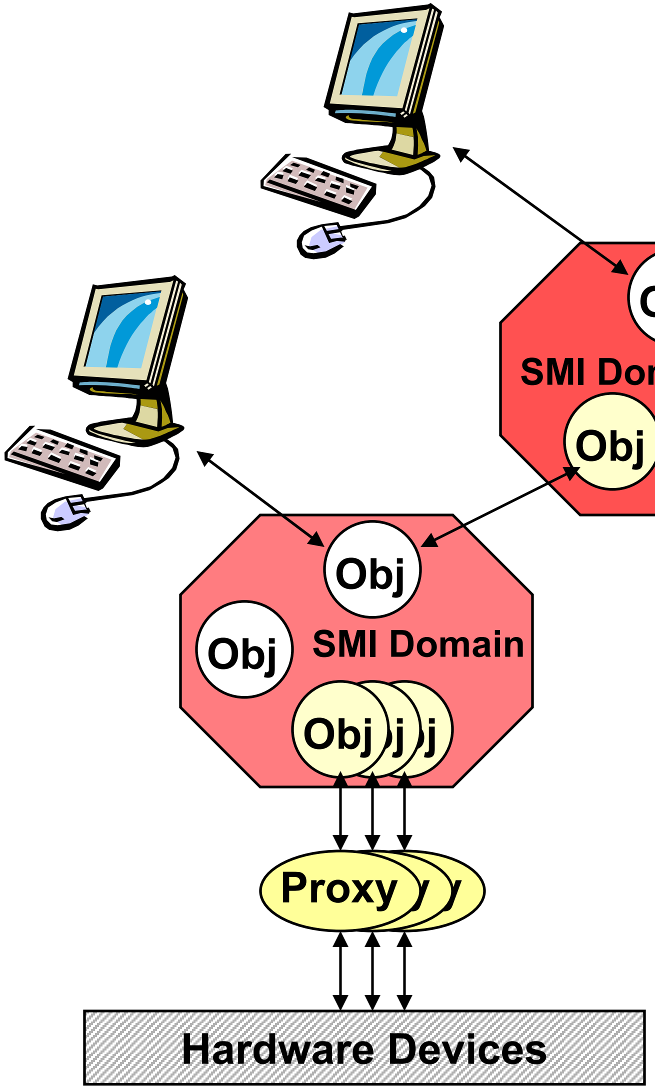
SMI++

- **Method**
  - Objects and Classes
    - 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
  - Expert System like rules
    - Allow Automation and Error Recovery

SMI++

- **Method (Cont.)**
  - SMI++ Objects can be:
    - Abstract (e.g. a Run or the DCS)
    - Concrete (e.g. a CAEN power supply or a tape)
  - Concrete objects interact with devices through associated processes - “proxies”
  - Logically related objects can be grouped inside “SMI Domains”

SMI++ Run-time Environment



- Device Level: Proxies
  - C, C++, PVSS ctrl scripts
  - drive the hardware:
    - deduceState
    - handleCommands
- Abstract Levels: Domains
  - Internal objects
  - Dedicated language
  - Implement the logical model
- 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
      - Actions can be triggered by logical conditions on the state of other objects

SML - The Language

- **An SML file corresponds to an SMI Domain. This file describes:**
  - The objects contained in the domain
  - For Abstract objects:
    - The states & actions of each
    - The detailed description of the logic behaviour of the object
  - For Concrete or External (Associated) objects
    - The declaration of states & actions

SML - example

```
class: HV
   state: NOT_READY /initial_state
      when (CAEN1 in_state ON) move_to READY
      action: GOTO_READY
         do SWITCH_ON CAEN1
         if (CAEN1 in_state ON) then
            move_to READY
         endif
         move_to ERROR
   state: READY
      when (CAEN1 in_state TRIP) do RECOVER
      action: RECOVER
         do RESET CAEN
         do SWITCH_ON CAEN
         …
      action: GOTO_NOT_READY
      …
   state: ERROR
      …
   state: TRIP
      …

object: MUON_HV is_of_class HV
```

```
class: CAEN /associated
   state: UNKNOWN /dead_state
   state: OFF
      action : SWITCH_ON
   state: ON
      action : SWITCH_OFF
   state: TRIP
      action : RESET
      …

object: CAEN1 is_of_class CAEN
```

SMI++ Declarations

- **Classes, Objects and ObjectSets**
- class: <class_name> [/associated]
  - <parameter_declaration>
  - <state_declaration>
    - <when_list>
    - <action_declaration>
      - <instruction_list>
  - …
- object: <object_name> is_of_class <class_name>
- objectset: <set_name> [{obj1, obj2, …, objn}]

SMI++ Parameters

- **<parameters>**
  - SMI Objects can have parameters, ex:
    - int n_events, string error_type
  - Possible types:
    - int, float, string
  - For concrete objects
    - Parameters are set by the proxy
      (they are passed to the SMI domain with the state)
  - Parameters are a convenient way to pass extra information up in the hierarchy

SMI++ States

- **state: <state_name> [/<qualifier>]**
  - **<qualifier>**
    - /initial_state
      For abstract objects only, the state the object takes when it first starts up
    - /dead_state
      For associated objects only, the state the object takes when the proxy or the external domain is not running

SMI++ Whens

- **<when_list>**
  - Set of conditions that will trigger an object transition. "when"s are executed in the order they are declared (if one fires, the others will not be executed).
  - state: <state>
    - when (<condition>) do <action>
    - when (<condition>) move_to <state>

SMI++ Conditions

- <condition>
  - Evaluate the states of objects or objectsets
    - (<object> [not_]in_state <state>)
    - (<object> [not_]in_state {<state1>, <state2>, …})
    - (all_in <set> [not_]in_state <state>)
    - (all_in <set> [not_]in_state {<state1>, <state2>, …})
    - (any_in <set> [not_]in_state <state>)
    - (any_in <set> [not_]in_state {<state1>, <state2>, …})
    - (<condition> and|or <condition>)

SMI++ Actions

- **action: <action_name> [(parameters)]**
  - If an object receives an undeclared action (in the current state) the action is ignored.
  - Actions can accept parameters, ex:
    - action: START_RUN (string run_type, int run_nr)
  - Parameter types:
    - int, float and string
  - If the object is a concrete object
    - The parameters are sent to the proxy with the action
  - Action Parameters are a convenient way to send extra information down the hierarchy

SMI++ Instructions

- **<instructions>**
  - <do>
  - <if>
  - <move_to>
  - <set_instructions>
    - insert <object> in <set>
    - remove <object> from <set>
  - <parameter_instructions>
    - set <parameter> = <constant>
    - set <parameter> = <object>.<parameter>
    - set <parameter> = <action_parameter>

SMI++ Instructions

- **<do> Instruction**
  - Sends a command to an object.
  - Do is non-blocking, several consecutive "do"s will proceed in parallel.
    - do <action> [(<parameters>)] <object>
    - do <action> [(<parameters>)] all_in <set>
    - examples:
      - do START_RUN (run_type = "PHYSICS", run_nr = 123) X
      - action: START (string type)
        - do START_RUN (run_type = type) EVT_BUILDER

SMI++ Instructions

- <if> Instruction
  - "if"s can be blocking if the objects involved in the condition are "transiting". The condition will be evaluated when all objects reach a stable state.
    - if <condition> then
      - <instructions>
    - else
      - <instructions>
    - endif

SMI++ Instructions

- **<move_to> Instruction**
  - "move_to" terminates an action or a when statement. It sends the object directly to the specified state.
    - action: <action>
      - ...
      - move_to <state>
    - when (<condition>) move_to <state>

Addressing Objects

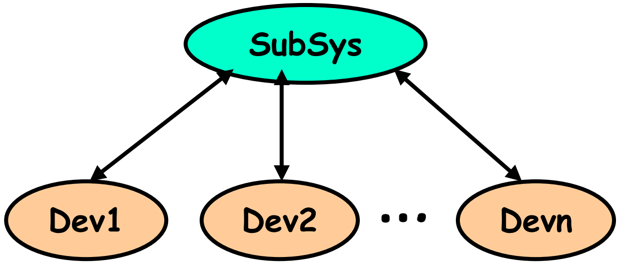
- Objects in different domains can be addressed by: <domain>::<object>

```
object: DET_CONTROL
   state: TEST_MODE
      when (LHC::STATE in_state PHYSICS) do PHYSICS
      action: PHYSICS
         do GOTO_READY MUON::MUON_HV
         …
   state: PHYSICS_MODE
      …
```

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

Handling Many Devices

- Object Sets



Where n = hundreds

```
class: DEV /associated
   ...
object: Dev1 is_of_class DEV
...
objectset: DEVICES {Dev1,Dev2,...}

object: SubSys
   state: READY
      when ( any_in DEVICES in_state ERROR) move_to ERROR
      action: START
         do START all_in DEVICES
         move_to RUNNING
      action: DISABLE_DEV(string device)
         remove &VAL_OF_device from DEVICES
   state: RUNNING
   ...
```

SMI++ tools

- **Tools for generating the run-time system**
  - **smiPreproc** (only on Linux for the moment)
    - Preprocessor: include, macros, etc.
  - **smiTrans** file.sml file.sobj
    - Parser and metafile generator
- **Tools for running the system**
  - **smiSM** domain_name file.sobj
    - SMI Engine/Scheduler
  - **smiGUI** domain_name (only on Linux for the moment)
  - **smirtl and smiuirtl libraries**

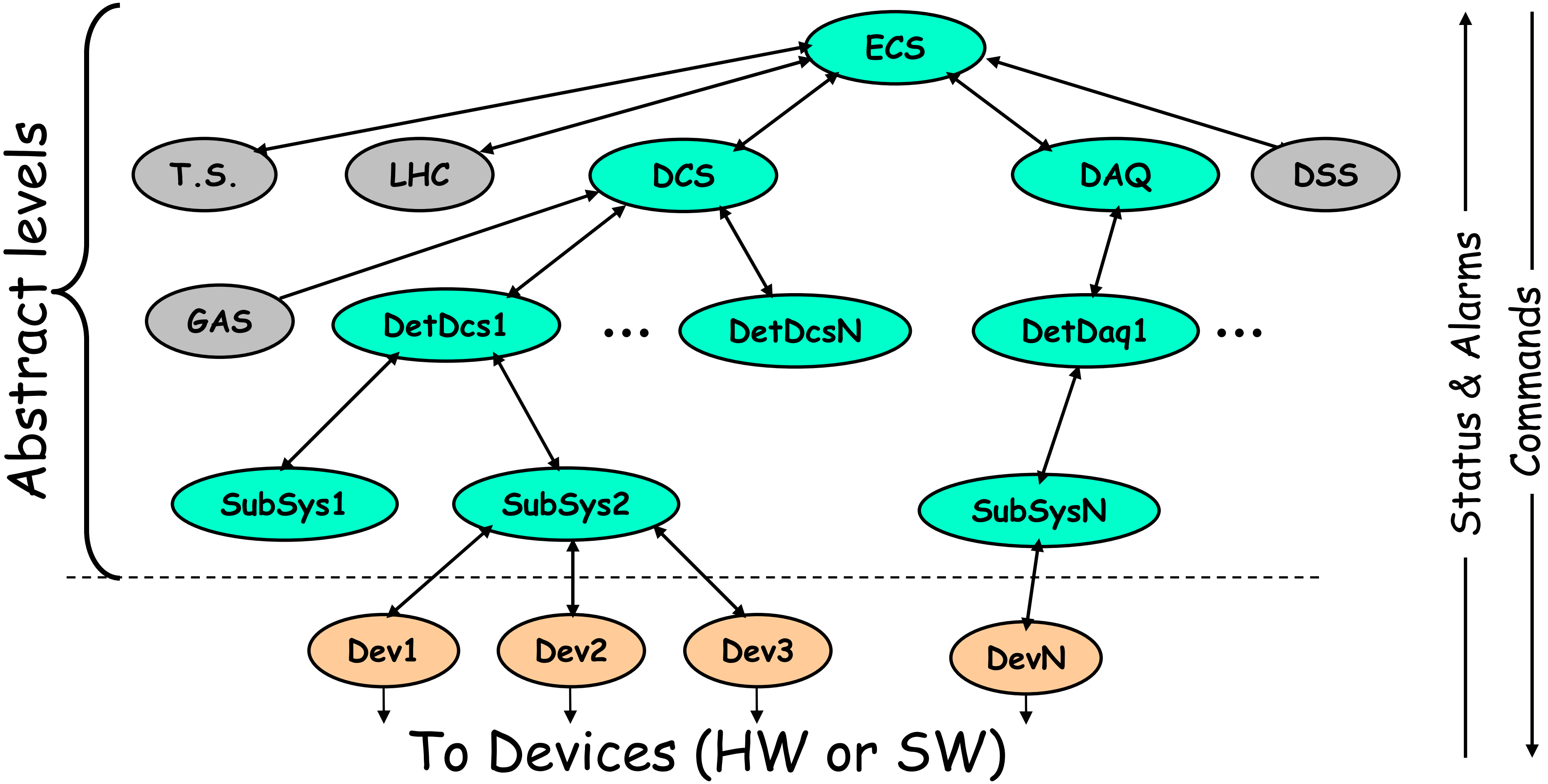
SMI++ Preprocessor

- File Include
    - #include "filename"
- Macros (recursive replacement)
        - .macro find_state(obj)
            - if (${obj} in_state ON) then
                - move_to ON
            - …
        - .endmacro
    - Inside an action:
        - find_state(ObjA)
        - .repeat find_state(ObjA, ObjB, ObjC)

SMI++ Libraries

- **Smirtl**
  - Available in C, and C++
  - To be used by Proxies
    - smiSetState
    - smiHandleCommand
- **Smiuirtl**
  - Also C and C++
  - To be used by clients (User interfaces)
    - handleStateChange
    - sendCommand

Framework Naming



Sys	Control Unit
Dev	Device Unit

Clara Gaspar, February 2004

Framework Definitions

- **Hardware Device**
  - a HV channel or an analog input organized by hardware type (CAEN, ELMB, etc.)
- **Logical Device**
  - a HV channel or an analog input organized by logical function (sub-detector, endcap, etc.)
- **Device Unit**
  - Implements the behaviour of a device or a group of devices, hardware (ex. CAEN create) or logical (ex. endcap temperature)

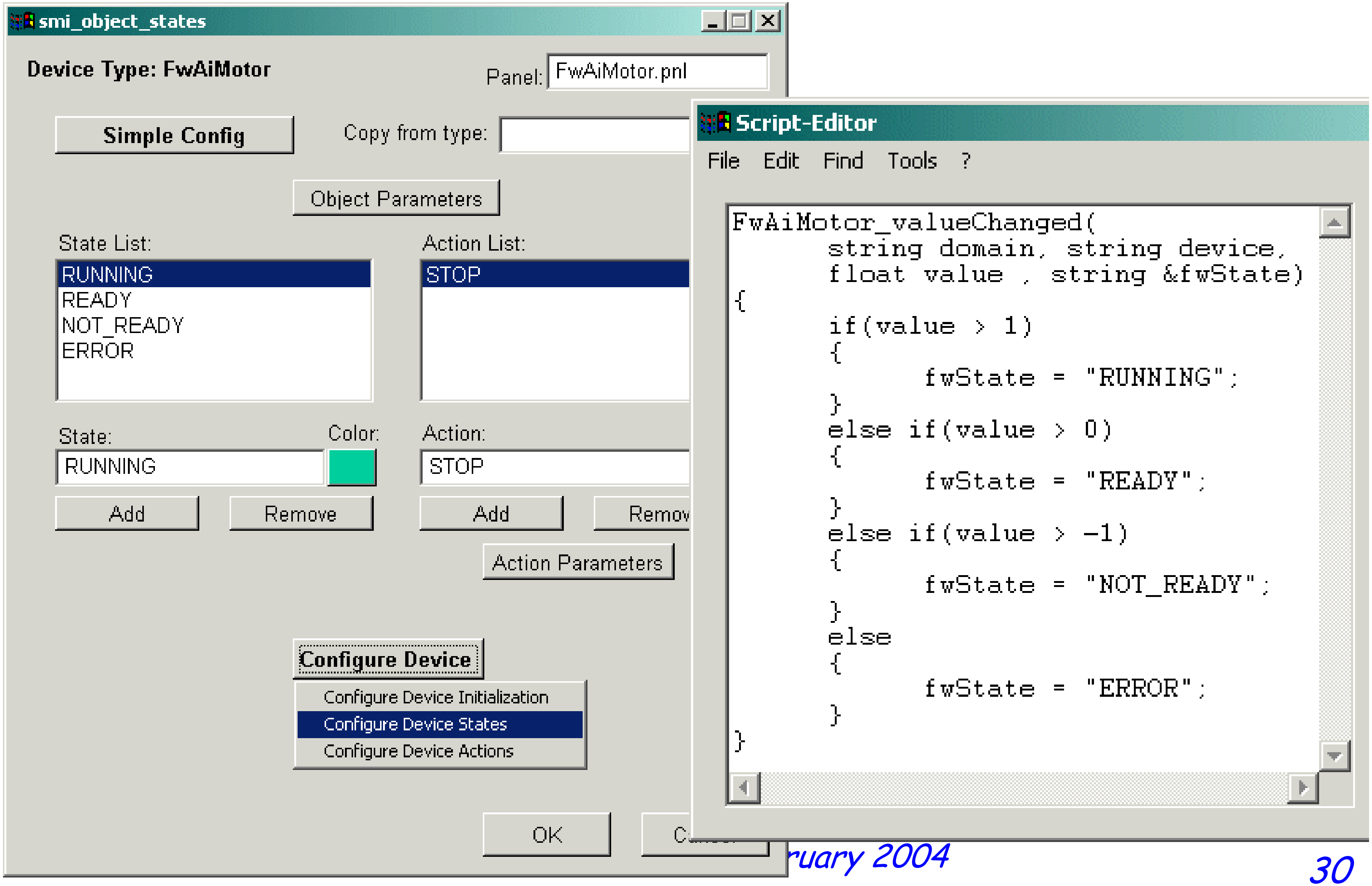
Framework Definitions

- **Control Unit**
  - Implements the behaviour of a SubSystem and its children (Device Units or other Control Units)
  - Implements the partitioning rules, i.e. knows how to include, exclude, etc. its children
    - Include/Exclude/Manual/Ignored
  - Implements Error Handling, i.e. can recover from errors coming from its children

FW <-> SMI++ Naming

- **Device Unit <-> Proxy**
  - Implements actions on the HW
  - Retrieves a state from the HW
  - A Device Unit is a PVSS Datapoint
    - For example of type:
      - fwAI -> if the DU corresponds to one analog input
      - fwCaenBoard -> if the DU corresponds to a CAEN board with its channels
      - fwNode -> if the DU corresponds to a logical node containing several devices (possibly of different types)
      - Any other DP type the user wants

DU Type Editor



smi_object_states

Device Type: FwAiMotor

Panel: FwAiMotor.pnl

Copy from type:

State List:
RUNNING
READY
NOT_READY
ERROR

Action List:
STOP

State: RUNNING

Color: 🟩

Action: STOP

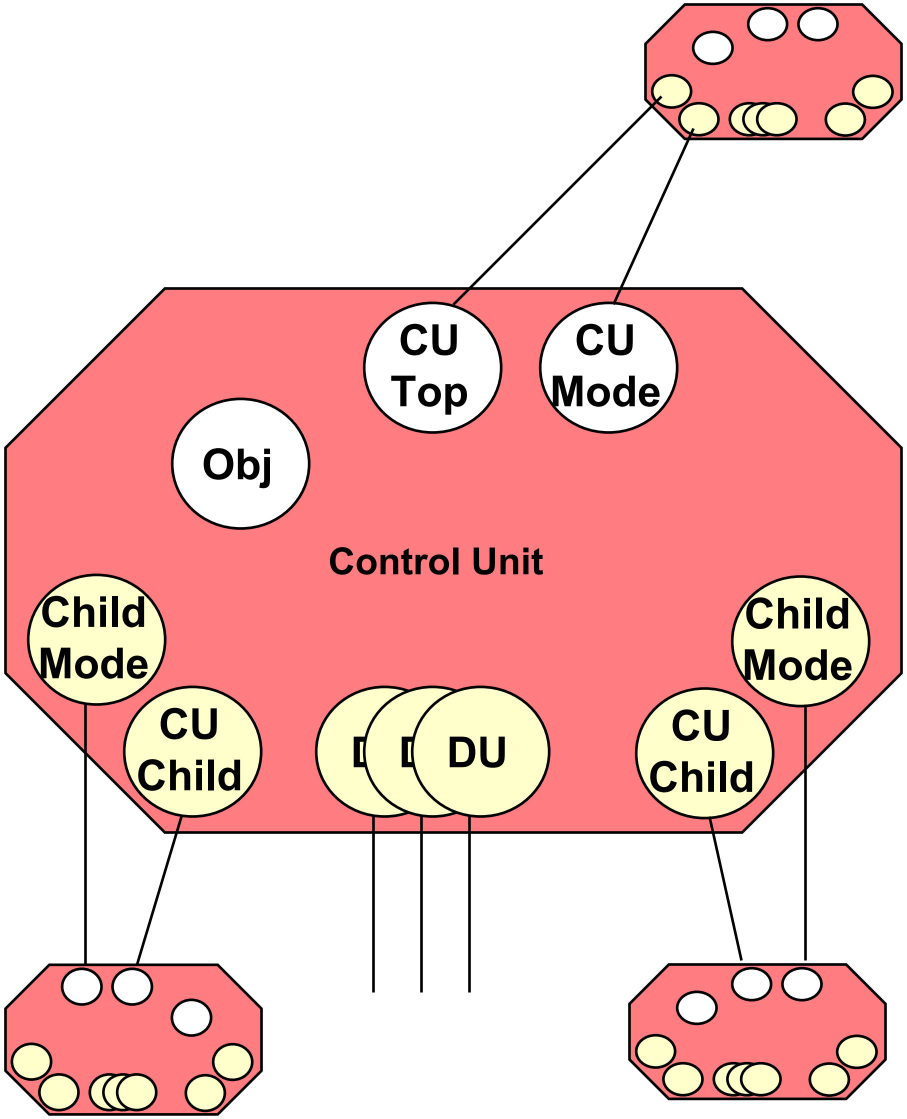
Script-Editor

```
FwAiMotor_valueChanged(
        string domain, string device,
        float value , string &fwState)
{
        if(value > 1)
        {
                fwState = "RUNNING";
        }
        else if(value > 0)
        {
                fwState = "READY";
        }
        else if(value > -1)
        {
                fwState = "NOT_READY";
        }
        else
        {
                fwState = "ERROR";
        }
}
```

FW <-> SMI++ Naming

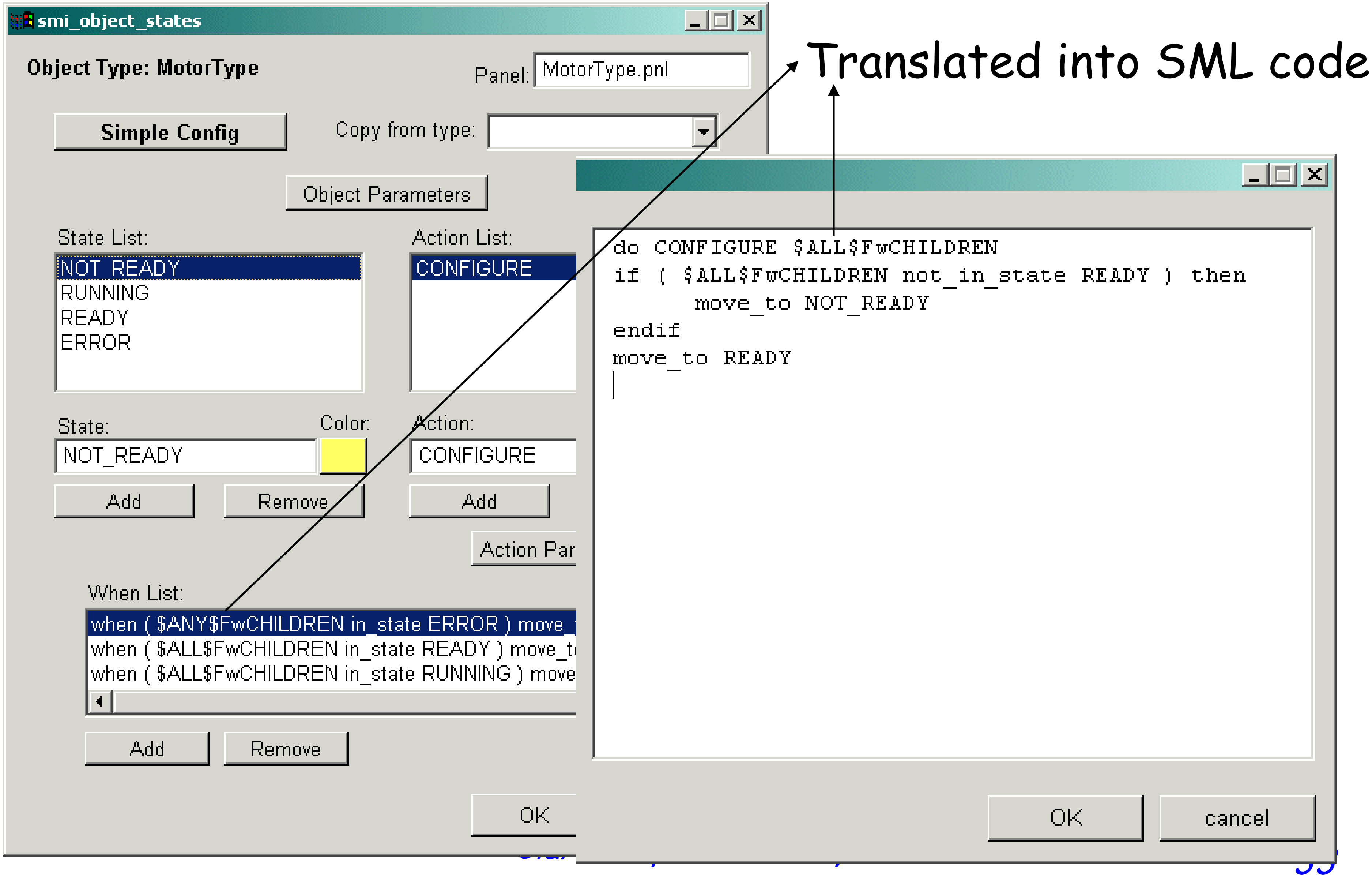
- **Control Unit <-> SMI Domain**
  - Containing:
    - One top-level Object (same name as CU)
      - Implementing the overall CU behaviour
        - keeps the overall state of the CU
        - Receives actions for this CU
    - Partitioning Objects (same for each CU)
      - Implementing the partitioning rules
    - Any other user defined Abstract Objects
    - Associated Objects for each of the children
      - DUs or other CUs
    - Children Objects in Sets for exclude/include

Control Unit SMI Domain



- Device Units are not partitionable (can not work in stand alone)
  -> No Mode Obj
- But they can be Disabled/Enabled

Object Type Editor



Translated into SML code

smi_object_states

Object Type: MotorType

Panel: MotorType.pnl

Copy from type:

State List:

- NOT_READY
- RUNNING
- READY
- ERROR

Action List:

- CONFIGURE

State: NOT_READY

Color: 🟨

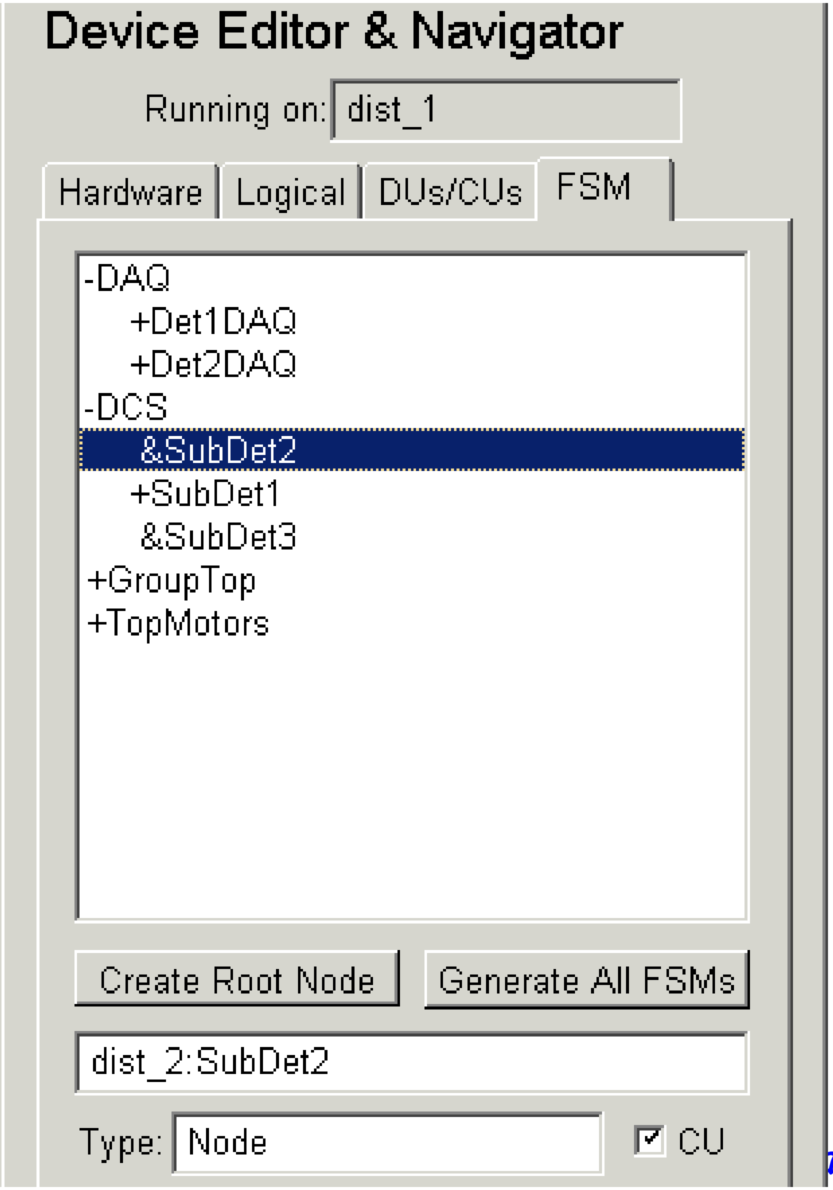
Action: CONFIGURE

When List:

```
when ( $ANY$FwCHILDREN in_state ERROR ) move
when ( $ALL$FwCHILDREN in_state READY ) move_t
when ( $ALL$FwCHILDREN in_state RUNNING ) move
```

```
do CONFIGURE $ALL$FwCHILDREN
if ( $ALL$FwCHILDREN not_in_state READY ) then
      move_to NOT_READY
endif
move_to READY
```

Hierarchy Building



Device Editor & Navigator

Running on: dist_1

```
-DAQ
   +Det1DAQ
   +Det2DAQ
-DCS
    &SubDet2
   +SubDet1
    &SubDet3
+GroupTop
+TopMotors
```

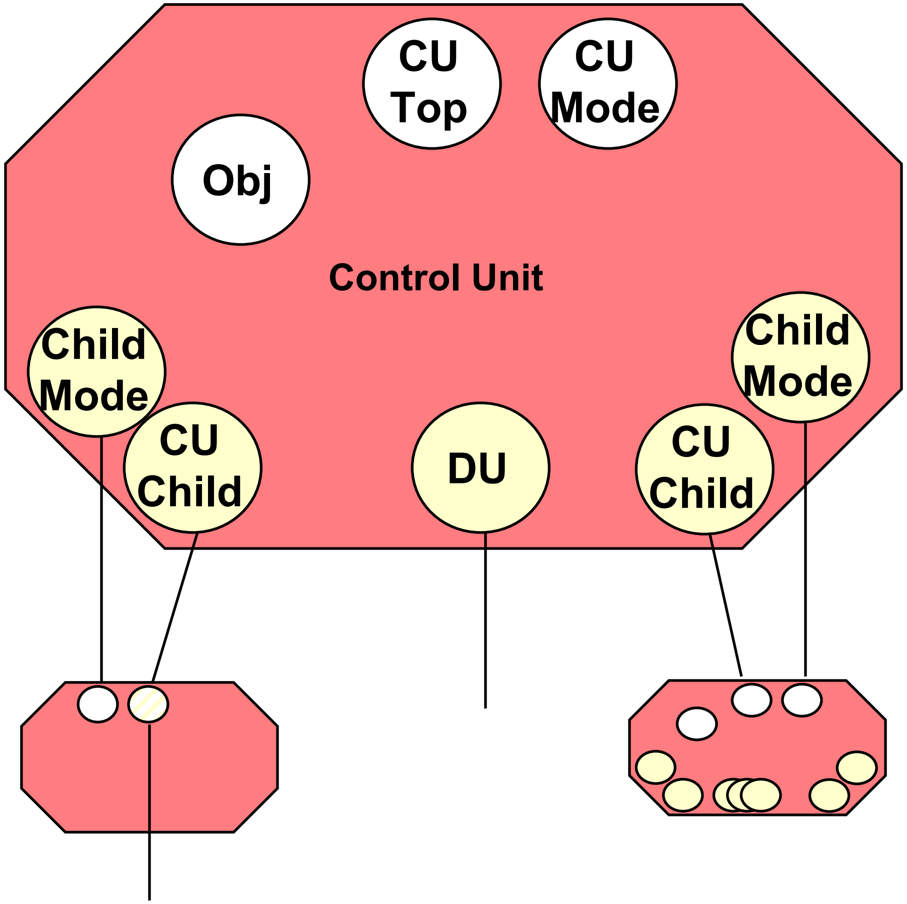
dist_2:SubDet2

Type: Node ☑ CU

- **Hierarchy of CUs**
  - "&" means reference to a CU in another system

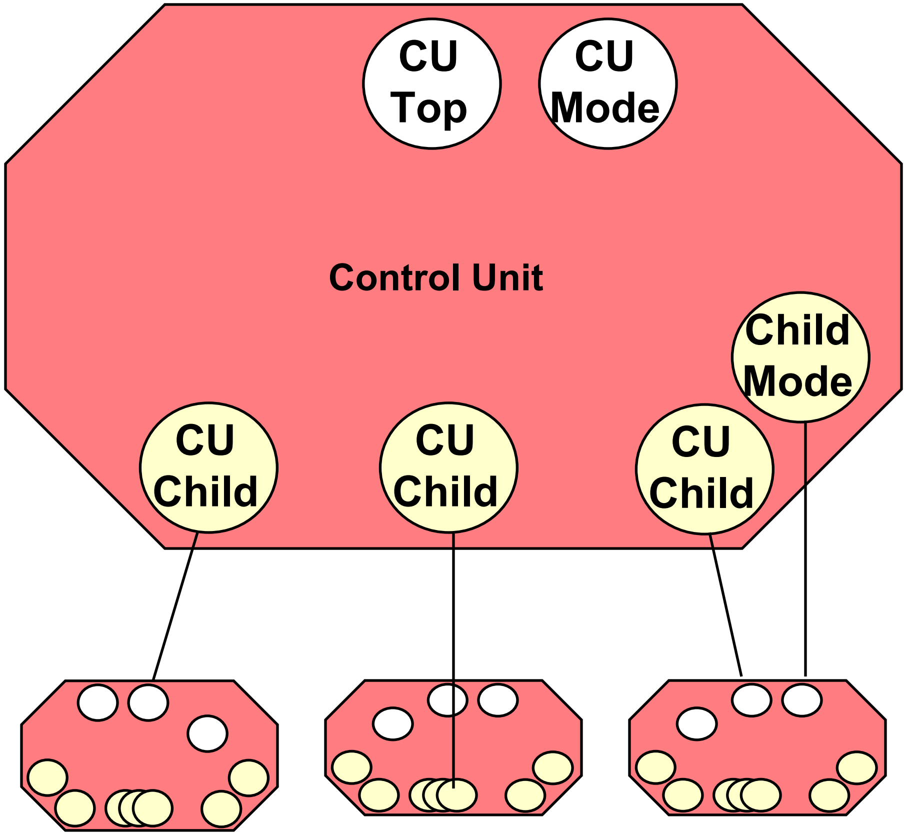
Hierarchy Building

- Create Root Node
- Add New Object
  - Not CU flag
- Add New Object
  - CU flag
- Add Device
  - Not CU flag
- Add Device
  - CU flag



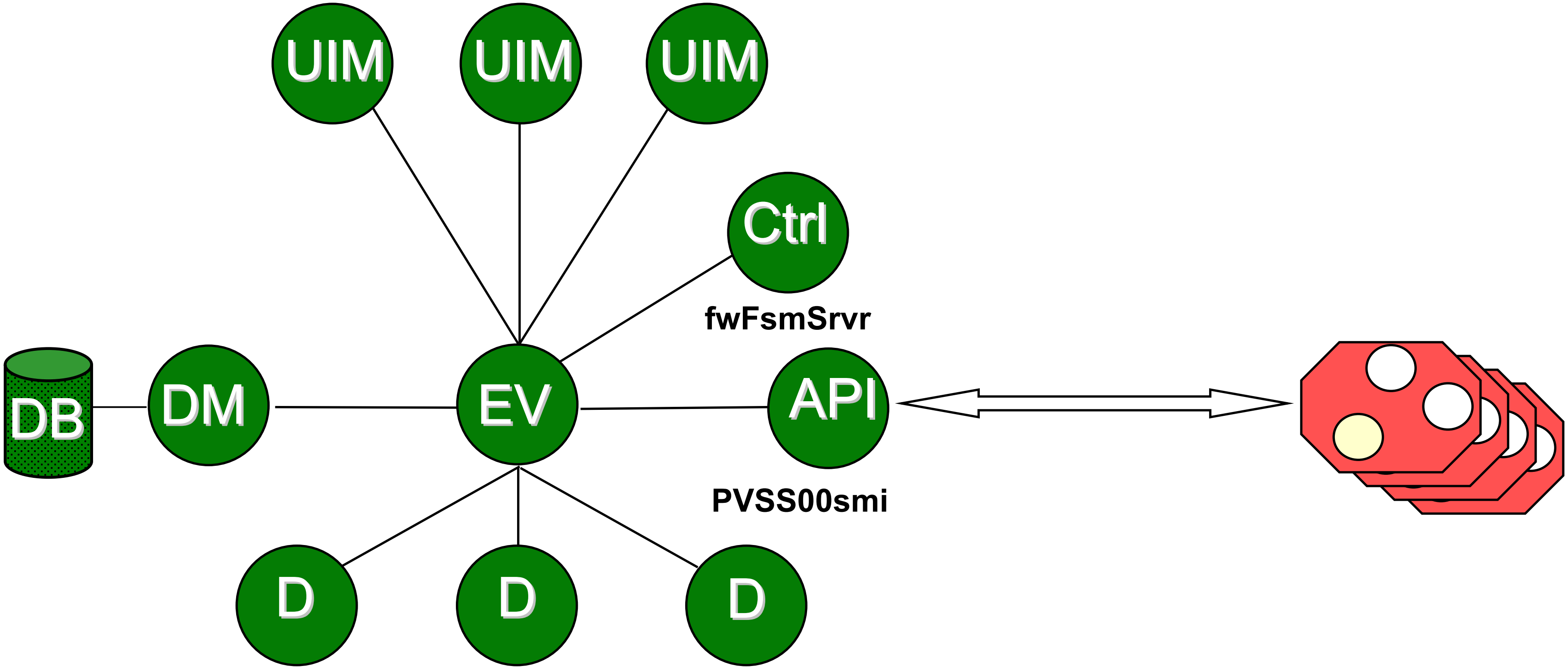
Hierarchy Building

- Create Root Node
- Add Object from FSM View
  - CU flag
- Add Object from FSM View
  - Not CU flag
- Add Device from FSM View
  - Not CU flag



Clara Gaspar, February 2004

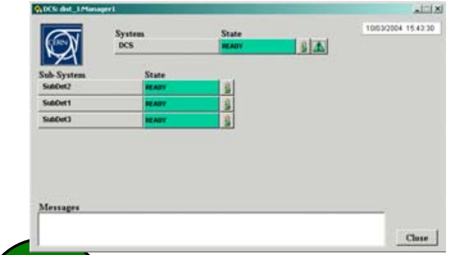
PVSS Integration



PVSS fwFsmSrvr

- **fwFsmSrvr ctrl script handles:**
  (one per PVSS System in the main node)
  - File Generation & Deletion
    - DU/Object Type
      - fsm files and DU ctrl scripts
    - Hierarchy Nodes
      - .sml files -> smiTrans -> .sobj files
  - Starting and Stopping smiSM Processes and PVSS00smi
  - Starting and Stopping DU ctrl scripts
    - a separate ctrl manager runs a thread per DU.
      - DU Handler

PVSS Integration



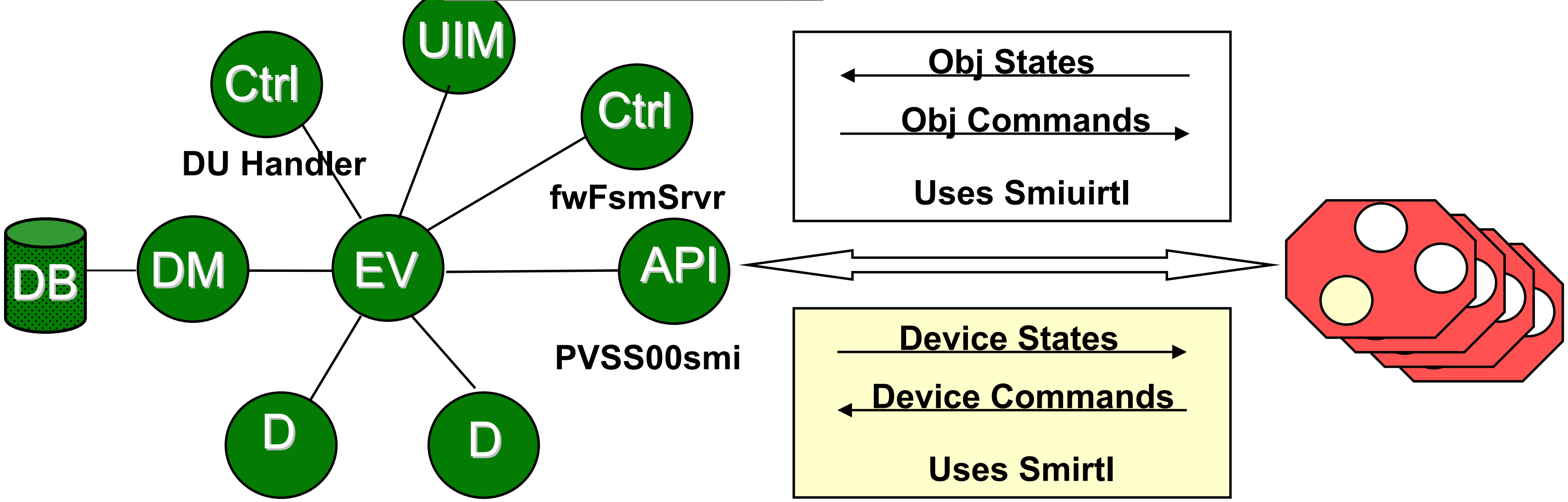
DCS: dist_1:Manager1

10/03/2004 15:43:30

System	State
DCS	READY

Sub-System	State
SubDet2	READY
SubDet1	READY
SubDet3	READY

Messages



Obj States

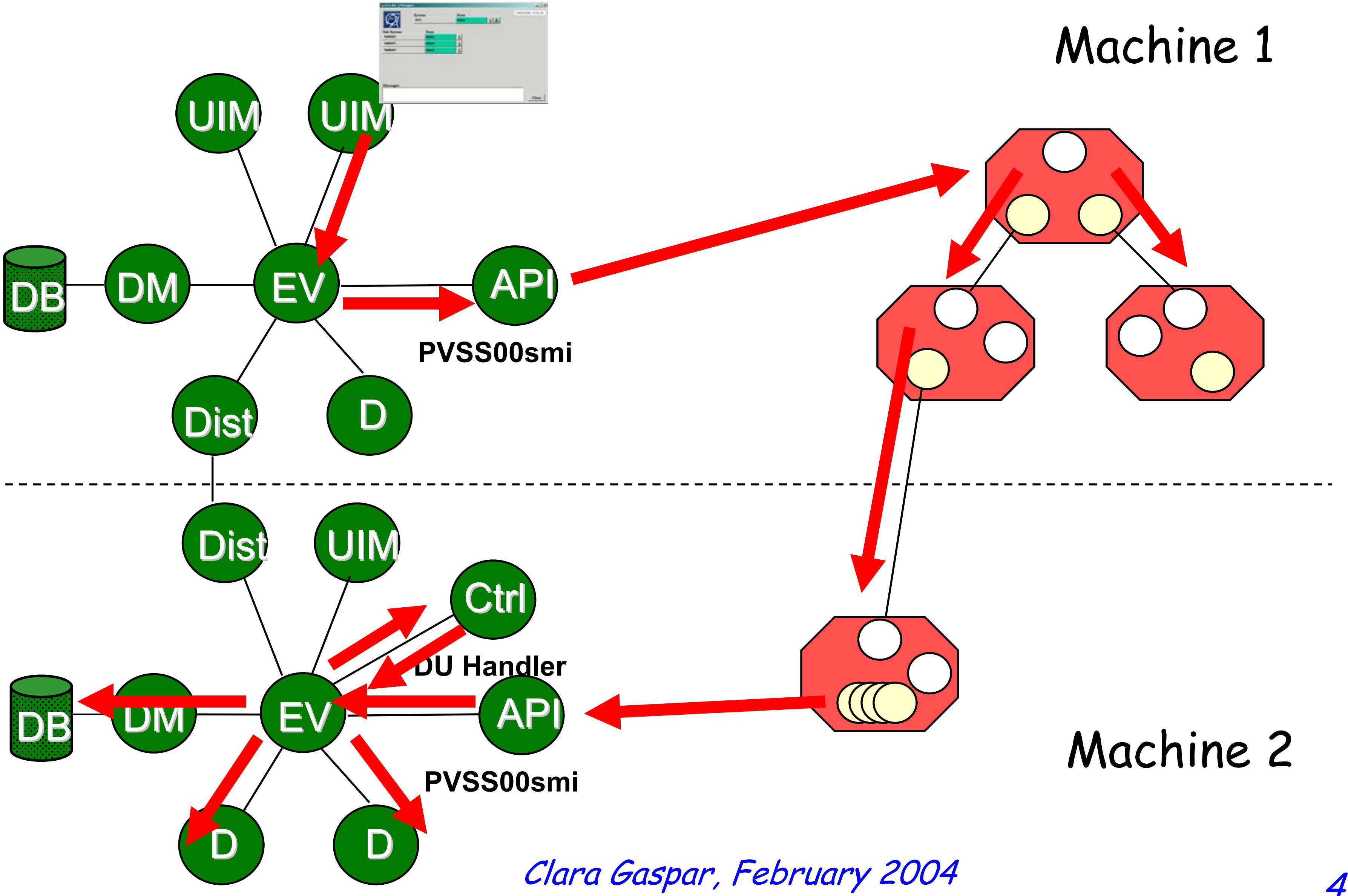
Obj Commands

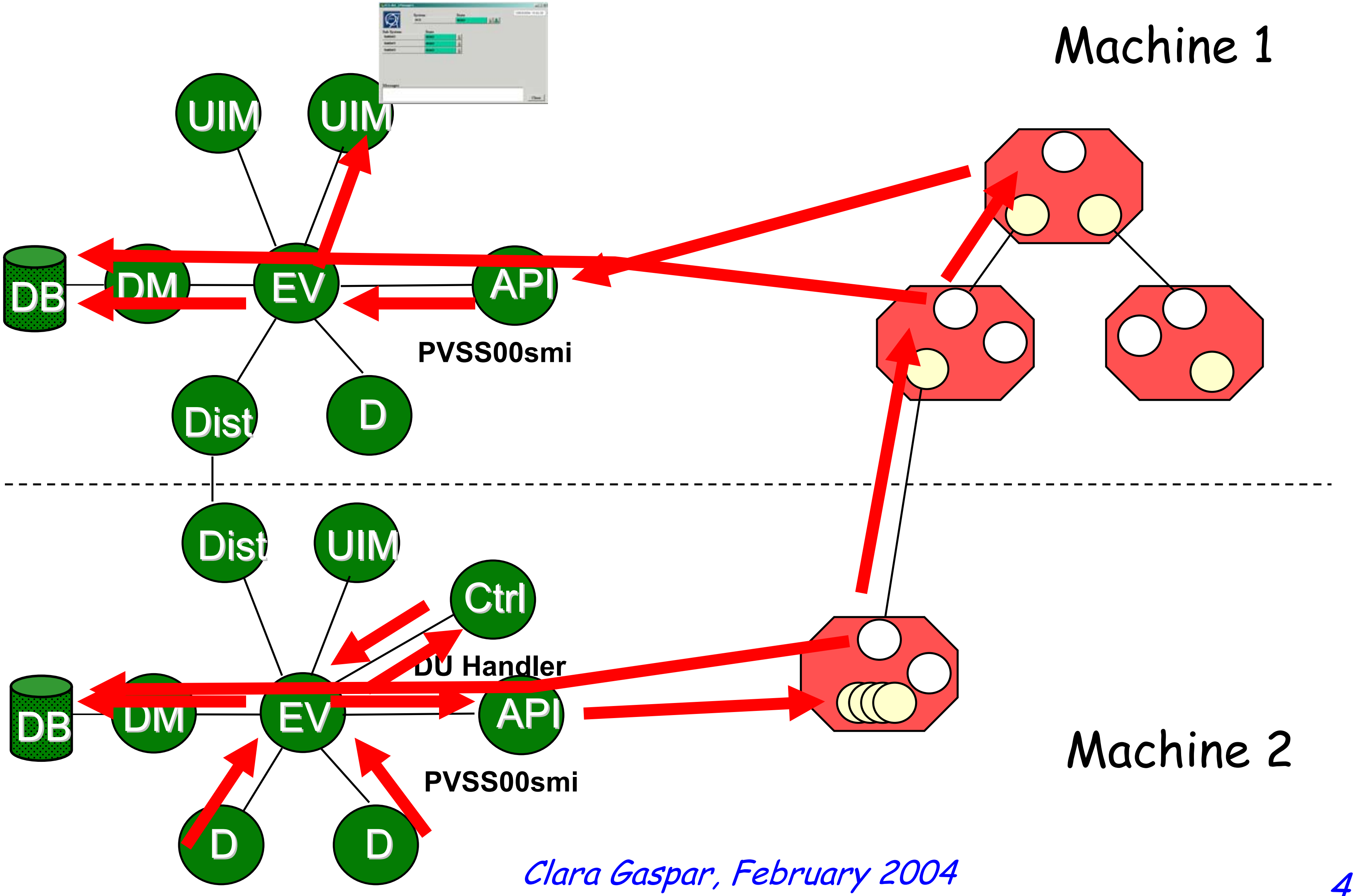
Uses Smiuirtl

Device States

Device Commands

Uses Smirtl


Machine 1
UIM
UIM
DB
DM
EV
API
PVSS00smi
Dist
D
Dist
UIM
Ctrl
DU Handler
DB
DM
EV
API
PVSS00smi
D
D
Machine 2


Machine 1
UIM
UIM
DB
DM
EV
API
PVSS00smi
Dist
D
Dist
UIM
Ctrl
DU Handler
DB
DM
EV
API
PVSS00smi
D
D
Machine 2

FwFSM Libraries

- **fwDU**
  - Library to be used inside DU ctrl scripts
    - setTimeout(int time, string newState)
    - setObjectParameters
    - getActionParameters
    - getDeviceAlarmLimits
- **fwCU**
  - Library for PVSS clients (user interfaces)
    - getObjectStates /connectObjectStates
    - sendCommands
    - Take/Return CU Tree

Some Notes on SMI Usage

- **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 SML code

Some Notes on SMI Usage

- **Error Recovery Mechanism**
  - Bottom Up
    - SMI Objects wait for command answers
      - Proxies/DUs should implement timeouts
  - Distributed
    - Each Sub-System recovers its errors
      - Each team knows how to recover local errors
  - Hierarchical/Parallel recovery
  - Can provide complete automation
    -> no need for an expert System

Some Notes on SMI Usage

- **Device Grouping/Granularity:**
  - Shall a DU be a single HV channel?
    - More intuitive and much easier to implement
    - Or shall a DU represent a group of HV channels?
      - Group channels to find a state in PVSS scripts
        - **Almost a repetition of SML code logic**
  - Current measurements:
    - 500 DUs in one CU
    - send command to all -> all change state -> CU changes state
    - took 5 seconds (but only 256 MB PC)
  - Recommendation is yes
    - But up to around 500/1000 DUs per CU

Future Developments

- **SML Language**
  - Parameter Arithmetics
    - set <parameter> = <parameter> + 2
    - if (<parameter> == 5)
  - wait(<obj_list)
  - for instruction
    - for (dev in DEVICES)
      - if (dev in_state ERROR) then
        - do RESET dev
      - endif
    - endfor

Future Developments

- **SMI++ tools**
  - Preprocessor on windows
  - Optimize performance
- **FwFSM Toolkit**
  - Make it easier for users to change SML code
  - Improve robustness and diagnosis
  - etc.