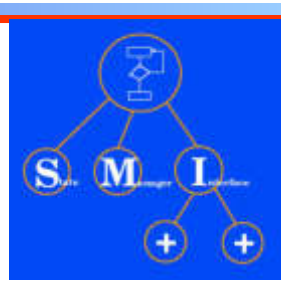


SMI++

A Tool for the Automation of large distributed control systems

Clara Gaspar, May 2010

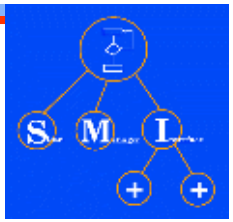
Outline

- Some requirements of large control systems
- SMI++
  - Methodology
  - Tools
- Example: Usage in LHC experiments
- Some important features
- Conclusions

Some Requirements

- Large number of devices/IO channels
  - ➡ Need for Distributed Hierarchical Control
    - De-composition in Systems, sub-systems, … , Devices
    - Local decision capabilities in sub-systems
- Large number of independent teams and very different operation modes
  - ➡ Need for Partitioning Capabilities (concurrent usage)
- High Complexity & Non-expert Operators
  - ➡ Need for Full Automation of:
    - Standard Procedures
    - Error Recovery Procedures
  - ➡ And for Intuitive User Interfaces

SMI++

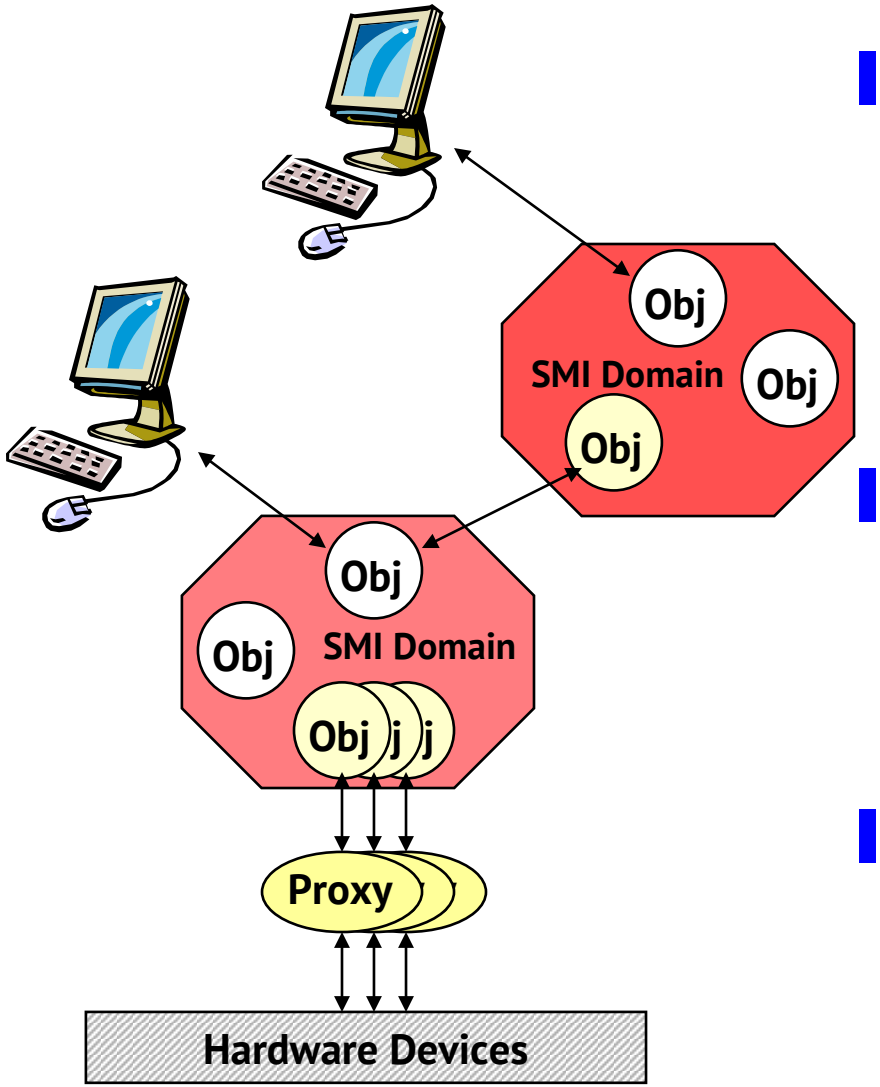


- **Method**
  - Classes and Objects
    - Allow the decomposition of a complex system into smaller manageable entities
  - Finite State Machines
    - Allow the modeling of the behavior of each entity and of the interaction between entities in terms of STATES and ACTIONS
  - Rule-based reasoning
  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

SMI++

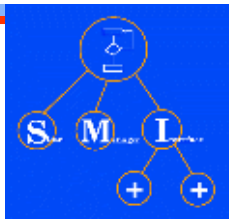
- **Method (Cont.)**
  - SMI++ Objects can be:
    - Abstract (e.g. a Run or the DCS System)
    - Concrete (e.g. a power supply or a temp. sensor)
  - Concrete objects are implemented externally either in "C", C++, or PVSS
  - Logically related objects can be grouped inside "SMI domains" representing 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++ - 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.
  - Synchronization and Sequencing
    - The user can also wait until actions finish before sending the next one.
  - 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
    when ( PS1 not_in_state ON ) move_to NOT_READY
    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
    when ( any_in PSS not_in_state ON ) move_to NOT_READY
    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:

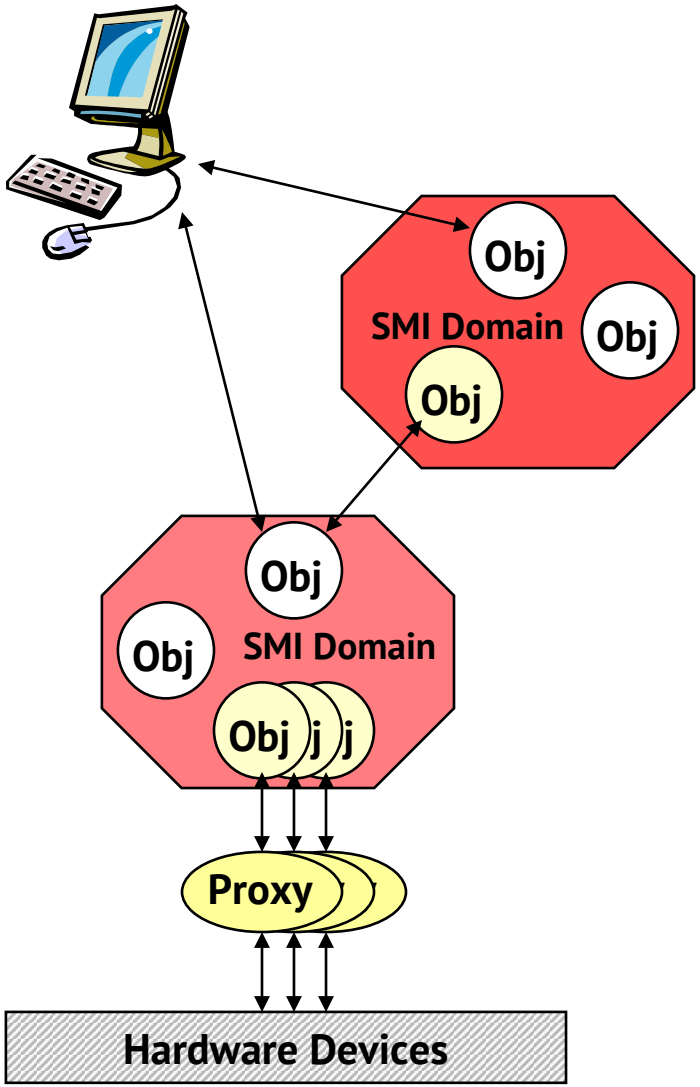
```
object: LHC::STATE /associated
  state: UNKNOWN /dead_state
  state: PHYSICS
  state: SETUP
  state: OFF
    ...
```

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

- 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

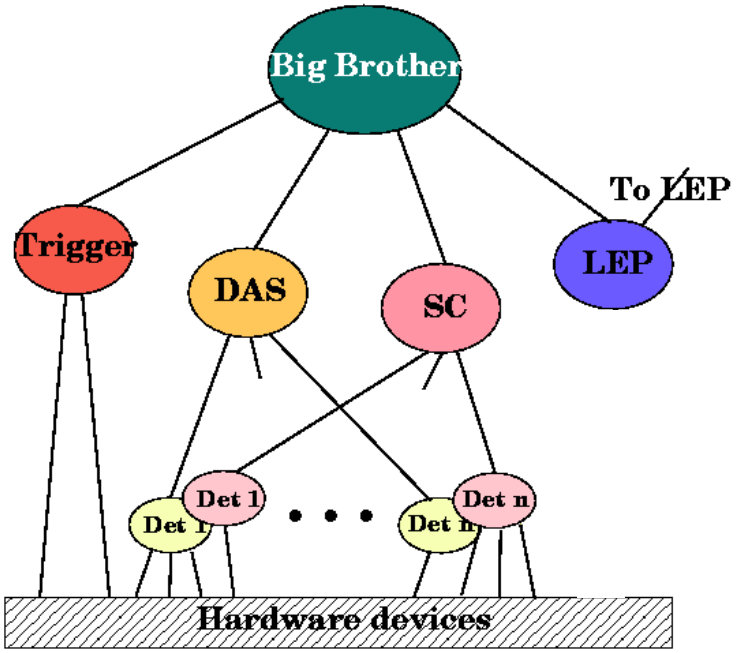


- Device Level: Proxies
  - C, C++, PVSS ctrl scripts
  - Use a Run Time Library: **smirtl**
    To Communicate with their domain
- Abstract Levels: Domains
  - A C++ engine: **smiSM** - reads the translated SML code and instantiates the objects
- User Interfaces
  - Use a Run Time Library: **smiuirtl**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

11

SMI++ History

- 1989: First implemented for DELPHI in ADA
  Thanks to M. Jonker and B. Franek in Delphi and the CERN DD/OC group (S. Vascotto, P. Vande Vyvre et al.)
  - DELPHI used it in all domains: DAQ, DCS, Trigger, etc.
    - A top level domain: Big-Brother automatically piloted the experiment
- 1997: Rewritten in C++
- 1999: Used by BaBar for the Run-Control and high level automation (above EPICS)
- 2002: Integration with PVSS for use by the 4 LHC exp.



➡ Has become a very powerful, time-tested, robust, toolkit

LHC Exp.: Framework

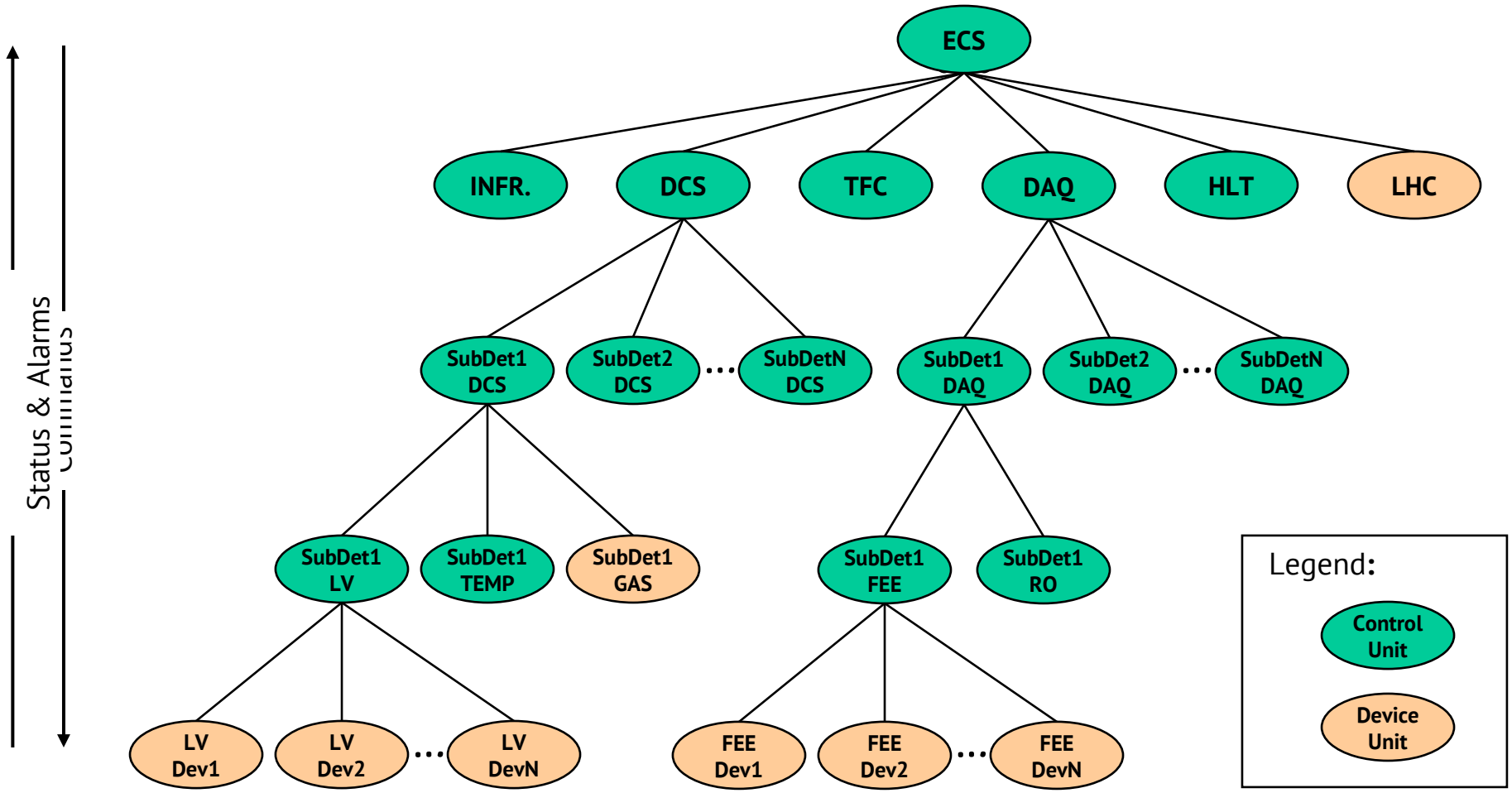
- **The JCOP Framework is based on:**
  - SCADA System - PVSSII for:
    - Device Description (Run-time Database)
    - Device Access (OPC, Profibus, drivers) + **DIM**
    - Alarm Handling (Generation, Filtering, Masking, etc)
    - Archiving, Logging, Scripting, Trending
    - User Interface Builder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

Device Units

Control Units

13

Ex: Generic LHC(b) Architecture



Device Units

Device Unit

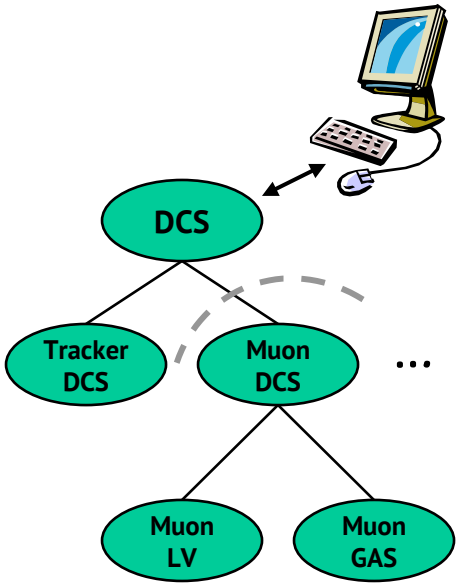
- **Provide access to "real" devices:**
  - The Framework provides (among others):
    - "Plug and play" modules for commonly used equipment. For example:
      - CAEN or Wiener power supplies (via OPC)
      - LHCb CCPC and SPECS based electronics (via DIM)
    - A protocol (DIM) for interfacing "home made" devices. For example:
      - Hardware devices like a calibration source
      - Software devices like the Trigger processes (based on LHCb's offline framework – GAUDI)
    - Each device is modeled as a Finite State Machine
      - ➡ Corresponds to an SMI++ Proxy

15

Hierarchical control

Control Unit

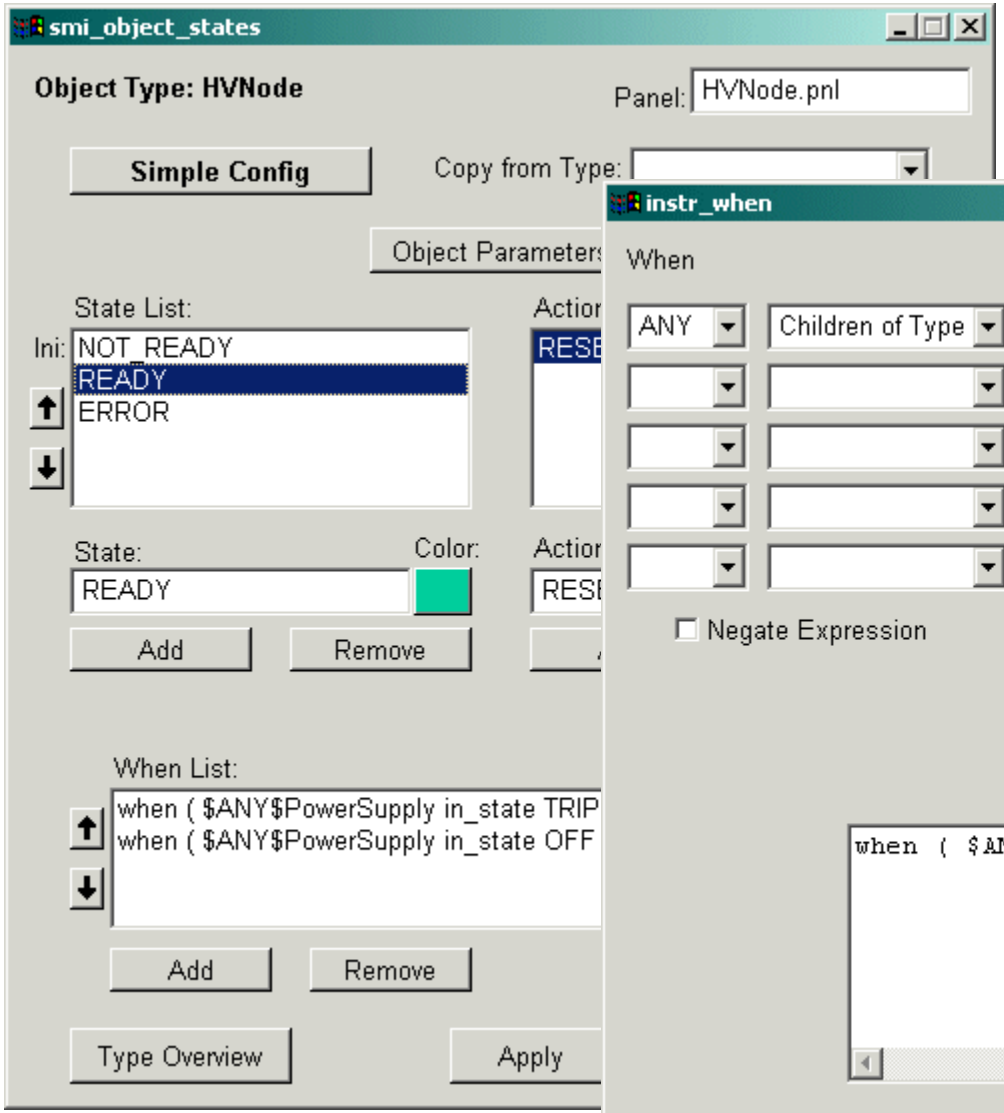
- **Each Control Unit:**
  - Is defined as one or more Finite State Machines
    - ➡ Corresponds to an SMI++ Domain
  - Can implement rules based on its children's states
  - In general it is able to:
    - Summarize information (for the above levels)
    - "Expand" actions (to the lower levels)
    - 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

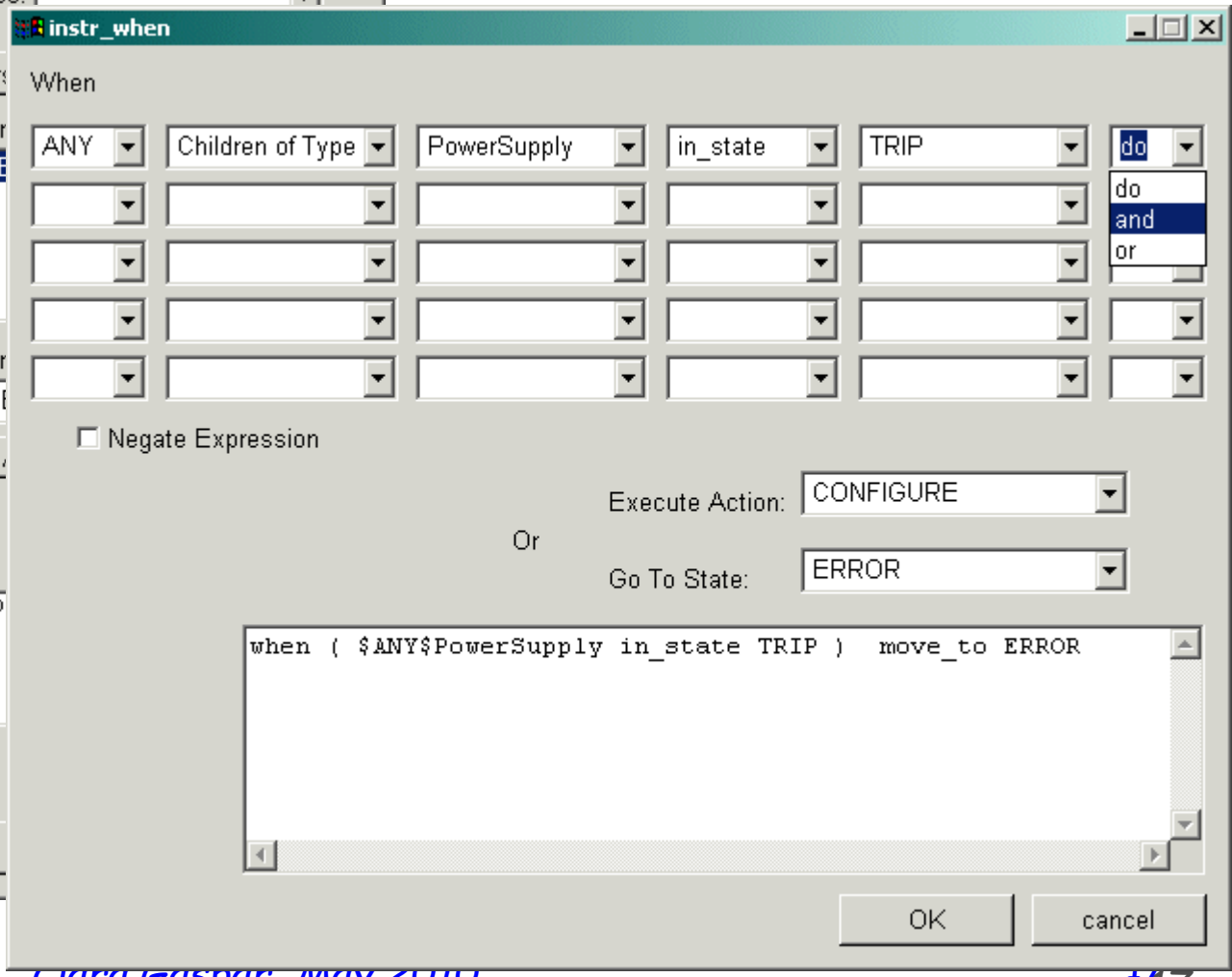


16

PVSS/SMI++ Integration

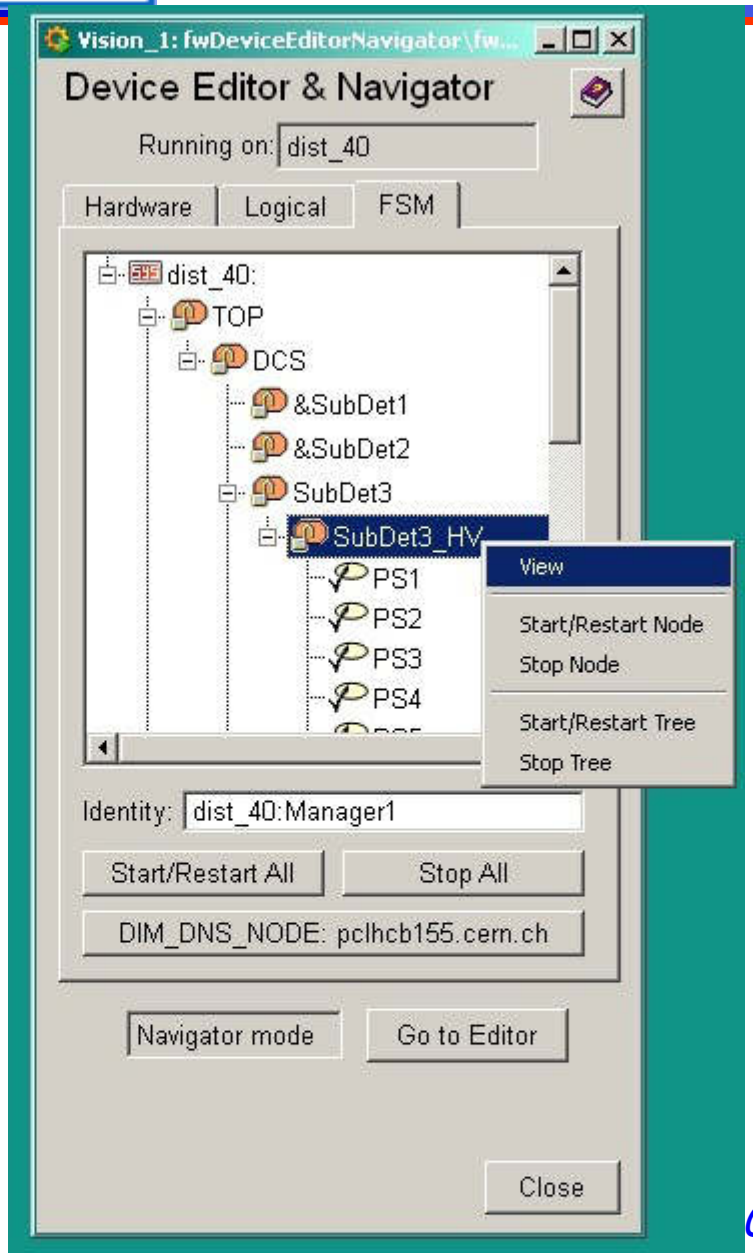
- Graphical Configuration of SMI++ Using PVSS





➡ Easy to learn SML

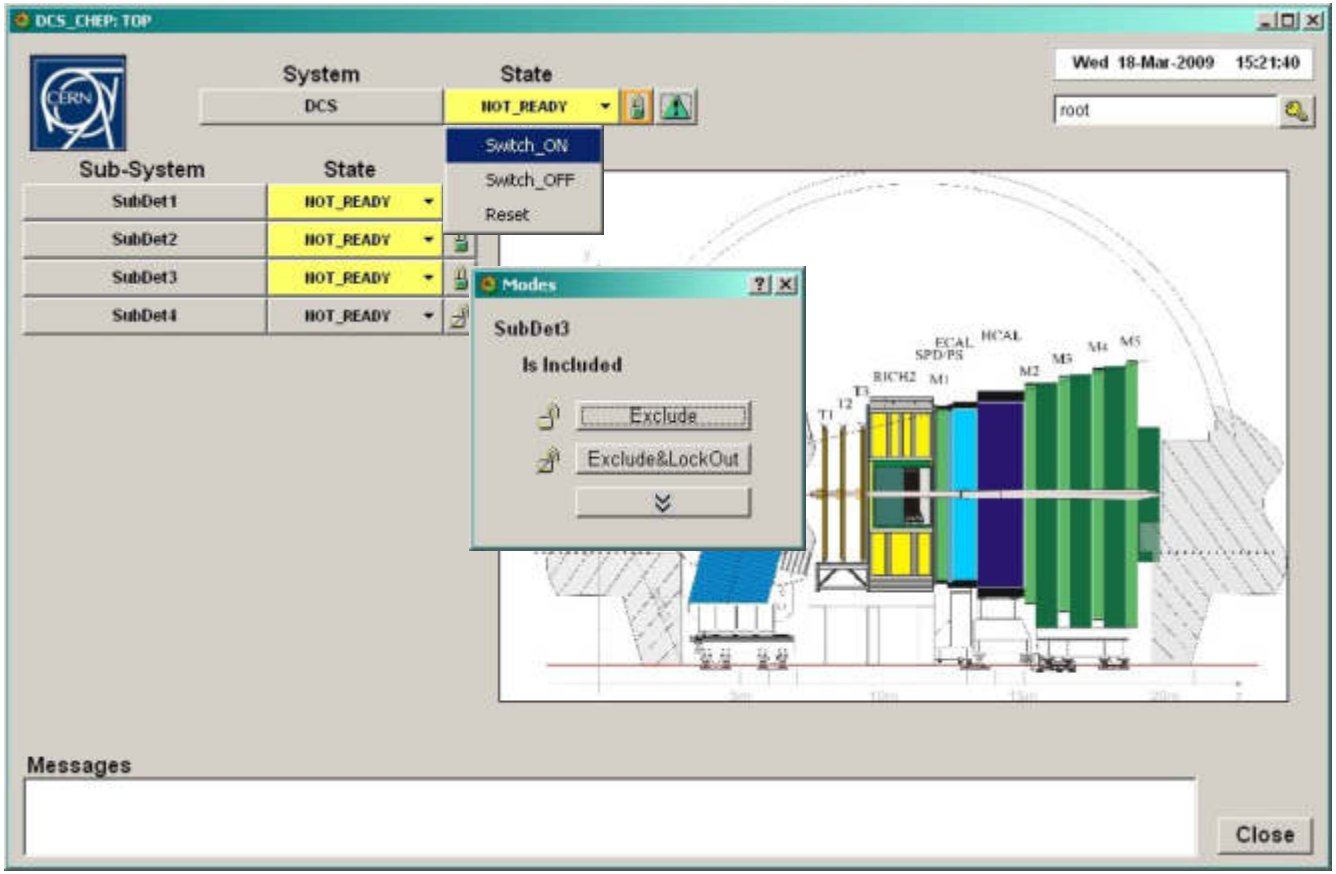
PVSS/SMI++ Integration



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

PVSS/SMI++ Integration

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



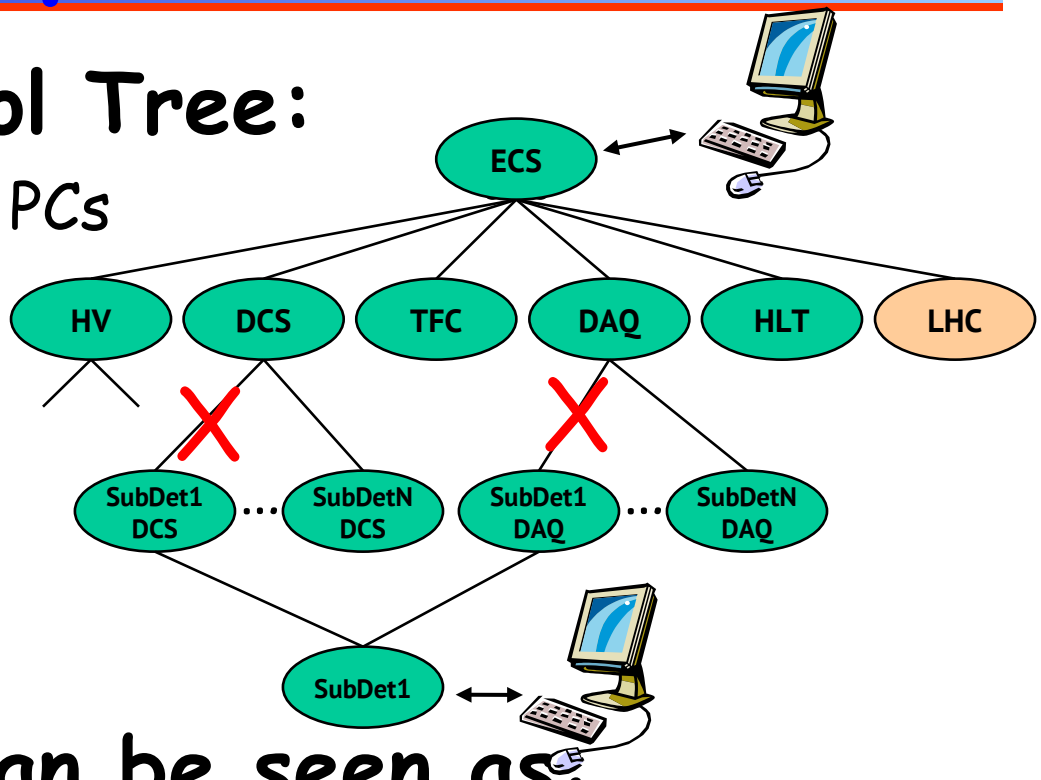
- Configurable
  User Panels
  and Logos
- "Embedded" standard partitioning rules:
  - Take
  - Include
  - Exclude
  - Etc.

"raw" SMI++ vs JCOP FW

- **SMI++ is a very powerful tool**
  - Its usage needs some training and experience
    - Learn the Language and the tools
    - Need to develop software using the libraries
      - To interface devices and to build User Interfaces
    - Define rules and guidelines for developers
- **While if using the JCOP FW:**
  - No need for software development
  - Graphic editing of the objects and their behaviour
  - All objects "inherit" the partitioning rules
  - JCOP provides training courses

LHCb Example: Run Control

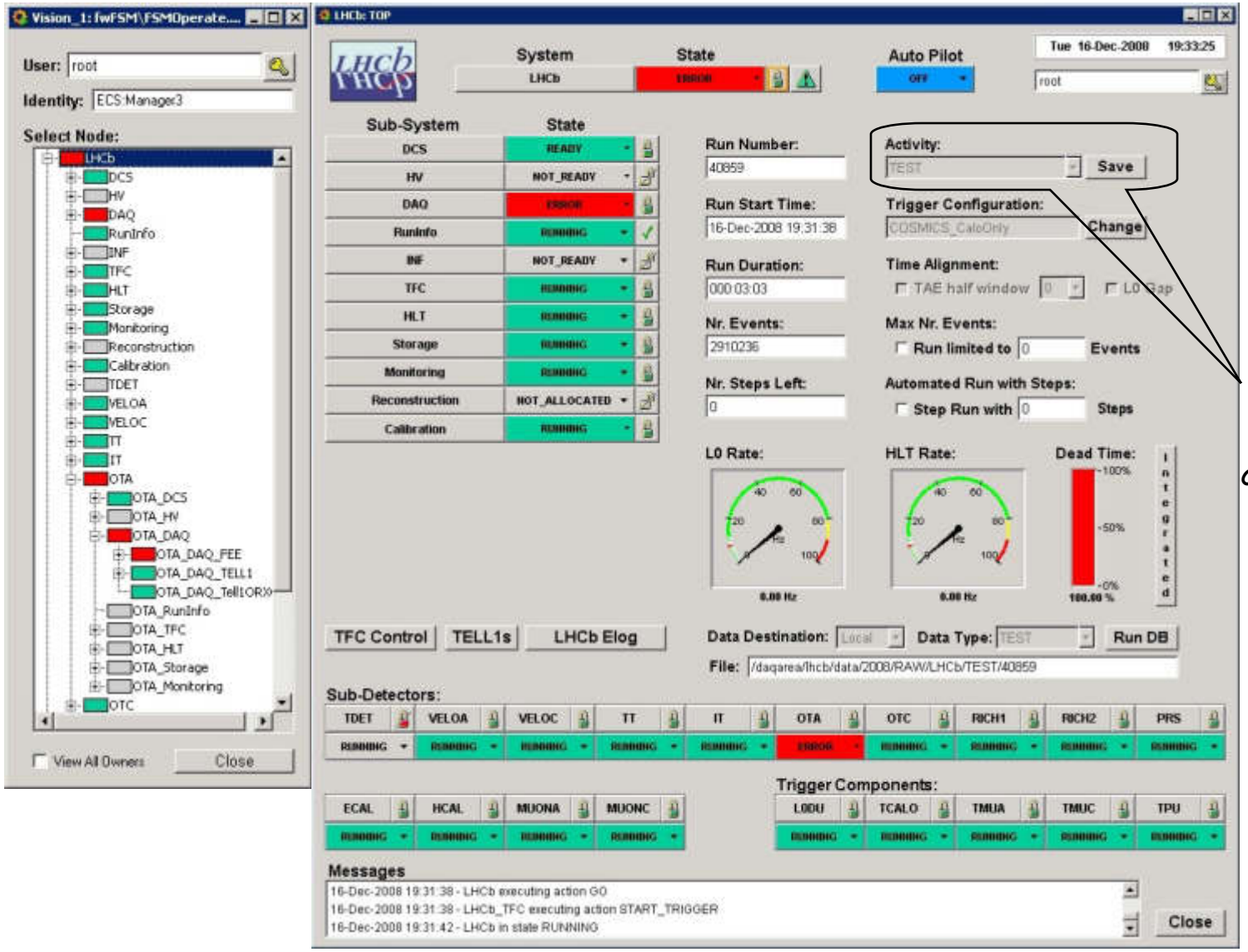
- **Size of the Control Tree:**
  - Distributed over ~150 PCs
    - ~100 Linux (50 for the HLT)
    - ~ 50 Windows
  - >2000 Control Units
  - >30000 Device Units



- **The Run Control can be seen as:**
  - The Root node of the tree
  - ➡ If the tree is partitioned there can be several Run Controls.

21

Run Control



- **Matrix**
  - Domain
  - X
  - Sub-Detector
- **Activity**
  - Used for Configuring all Sub-Systems
  - Accepts command parameters
- **SMI++**

Features of SMI++

- **Task Separation:**
  - SMI Proxies execute only basic actions – Minimal intelligence
    - Good practice: Proxies know "what" to do but not "when"
  - SMI Objects implement the logic behaviour
  - Advantages:
    - Change the HW
      -> change only the Proxy
    - Change logic behaviour
      sequencing and dependency of actions, etc
      -> change only SMI rules

23

Features of SMI++

- **Sub-system integration**
- **SMI++ allows the integration of components at various different levels:**
  - Device level (SMI++ All the way to the bottom)
    - Each Device is modeled by a Proxy
  - Any other higher level (simple SMI++ interface)
    - A full Sub-system can be modeled by a Proxy
    - Examples:
      - The Gas Systems (or the LHC) for the LHC experiments
      - Slow Control Sub-systems (EPICS) in BaBar

Features of

- Distribution and Robu
  - SMI Proxies and SMI
    distributed over a larg
    heterogeneous machin
  - If any process dies/cr
    - Its "/dead_state" is pr
  - When a process restarts (even on a different machine)
    - All connections are dynamically re-established
    - Proxies should re-calculate their states
    - SMI Objects will start in "/initial_state" and can recover their current state (if rules are correct)

```
class: HighVoltage
  state: NOT_READY /initial_state
    when ( any_in PSS in_state TRIP ) move_to ERROR
    when ( all_in PSS in_state ON ) move_to READY
    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
    ...
  state: ERROR
    ...

object: SubDetHV is_of_class HighVoltage
```

Features of SMI++

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

26

Conclusions

- **SMI++ is:**
  - A well tested, and very robust tool
  - Not only a Finite State Machine toolkit
  - But has also "Expert System" capabilities
    - Advantage: Decentralized and distributed knowledge base
  - Using the JCOP FW instead of directly SMI++ has many advantages…

Spare slides

28

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

30

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

31

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>

32

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>)

33

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

34

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>

35

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

36

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

37

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>

38

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

39

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