



Online syntax highlighting and checker for Control Unit Type of Finite State Machine

August 2016

Author:
Francesca Cecilia Schiavi

Supervisors:
Jean-Charles Tournier
Manuel González Berges

CERN openlab Summer Student Report 2016

Abstract

CERN describes the Detector Control Systems (DCSes) of the experiments of the Large Hadron Collider using finite state machines organised in a hierarchical directed acyclic graph structure. These systems are huge and impossible to understand only with human intervention, so computer checking is needed. Before the introduction of the tool described in this document, the analysis of the code controlling the DCSes was performed few times a week, introducing problem solving delay. The tool proposed allows an online time checking of the detectors' Control Units, thanks to the integration with the WinCCOA panel. WinCCOA is a SCADA (Supervisory Control and Data Acquisition) system tool that is used to develop the Control System applications at CERN.

Chapter breakdown:

Chapter 1: introduction to the environment, explaining why the project has been proposed and what is the aim of the tool in solving current problems.

Chapter 2: description about the framework that manage the control environment at CERN, and main concepts of the language that must be controlled by the tool.

Chapter 3: general illustration of the employed tools, ACE and PEG.js

Chapter 4: description on how the work has been implemented, from the setting up of the work environment, to the syntax rules and error reporting

Chapter 5: future development of the tool

Chapter 7: conclusions

Table of Contents

Abstract.....	2
1 Introduction	5
1.1 Chapter Breakdown	6
2 SMI++	6
2.1 State Manager Overview	6
2.2 SMI.....	7
2.2.1 Object model.....	8
2.2.2 Object communication	8
2.3 Design the control system.....	8
2.4 SML.....	9
3 Tools.....	10
3.1 ACE	10
3.1.1 Set up environment.....	10
3.1.2 ACE's rules	11
3.2 PEG.js.....	13
3.2.1 What is a parser.....	13
3.2.2 Set up environment.....	13
3.2.3 Generate the parser.....	14
3.2.4 PEG.js's rules	14
4 Development and Implementation	15
4.1 ACE	15
4.1.1 Mode Rules.....	15
4.2 PEG.....	17
4.2.1 Creation of Syntax Rules On-line	17
4.2.2 Generation of the Parser	19

4.2.3	Input File Checking	19
4.2.4	Error reporting.....	19
5	Future Scope	20
5.1	Integration with WinCCOA	20
5.2	Create Final State Machine Visualization	20
6	Conclusions	22
	Appendix A.....	22
	CLASS declaration	22
	OBJECT declaration.....	23
	STATE declaration.....	23
	ACTION declaration	23
	WHEN statement.....	24
	DO statement.....	24
	IF statement.....	24
	Appendix B.....	24
	References.....	27

1 Introduction

The Large Hadron Collider at CERN is a circular particle accelerator designed for the purpose of colliding particles in its experiments. In all experiments, a Detector Control System (DCS) monitors and controls environment variables such as voltage, temperature and humidity within the detector. It consists of many pieces of hardware that measure and control over a million parameters concerning these environment variables. All these sensors cannot be monitored by operators, so a DCS consists of software designed to gather information from its hardware sensors in order to summarise it to human operators and send commands to the hardware.

The architecture of a control unit of the DCS consists of nodes organized in a hierarchical structure (Figure 1.1).

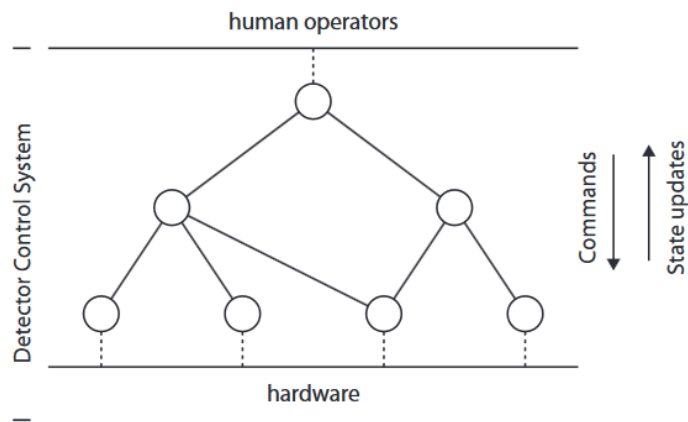


Figure 1.1: Architecture of a Detector Control System

Nodes gather the states of their children, use that information to move to a new state and notify their parents about the new state. Nodes also receive and send commands to trigger activity in their children.

The problem of controlling the various activities that constitute the data-acquisition and/or the slow control system of an experiment is a difficult, mainly because of the complexity of the required operations. In order to cope with the complexity of the online control system before the LHC-era the DELPHI experiment at CERN developed a new concept for the coding of the control logic. The adopted approach to the control problem is based on the State Manager (SM): the control aspects of an experiment are simulated by a finite state machine, which represents the behavioural model of the equipment under control. The various procedures necessary to run the experiment are executed by the SM which operates the corresponding state transitions on the state machine. Later this concept has been extended and redesigned using object-oriented techniques in SMI++ for the BaBar experiment at SLAC and is now used in all LHC experiments.

Nodes are modelled in the Finite State Machine (FSM) language, being an abstraction of the State Manager Language (SML), which runs on the State Management Interface (SMI++) framework that is part of the Joint Control Project (JCOP) framework.

Due to the sheer sizes, getting and maintaining complete overview of a DCS is very hard if not impossible for any human. This makes system maintenance and error discovery hard to do by hand. In 2012, the CMS experiment developed a set of tools (see reference [2] Leemans, 2012) that periodically check the semantic of the code of all the CU types defined in a system, but the delay between the modification of the control unit and the feedback to developers is too long.

To shorten the feedback given to the developers, this project aims to create an editor with syntax validation and highlighting for the development of FSM CU type. The editor has been created in JavaScript to allow future integration with WinCCOA panel. The syntax validation has been performed with PEG.js parser and the syntax highlighting has been handled with ACE.

1.1 Chapter Breakdown

Chapter 1: introduction of the problem's background and motivation behind the implementation of the editor.

Chapter 2: illustration of State Management Interface framework and State Manager Language, as well as a brief description on the State Machine concept.

Chapter 3: general explanation on the tools used to implement the project, highlighter ACE and parser PEG.js

Chapter 4: description of the development model and the implementation for the completion of project.

Chapter 5: point out possible future usage of the tool and an initial draft of integration with WinCCOA.

Chapter 6: summary of result and conclusion.

2 SMI++

SMI++ [1] is a framework for designing and implementing distributed control systems based on the original State Manager (SM) concept.

2.1 State Manager Overview

The State Manager (SM) is a computer-based control system for the experiment. The experiment (a set of hardware devices and software programs) is seen by the SM exclusively through computer processes, dedicated to one specific activity, called associated processes or proxies (Figure 2.1). The basic action by which the SM can control activities in the experiment is the exchange of messages with the associated processes, allowing the SM to have a unique interface with the outside world.

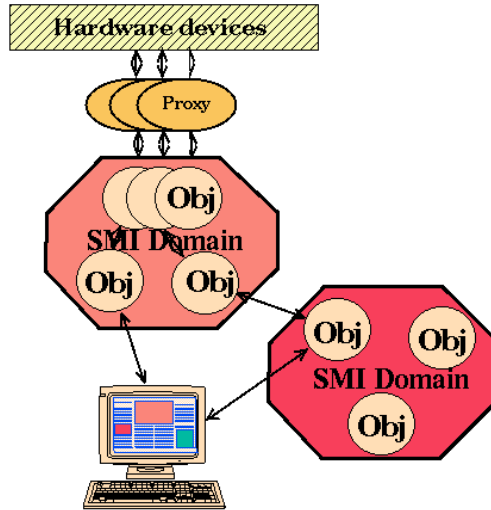


Figure 2.1 SMI environment (Source: Franek, Gaspar, "SMI++ User Manual")

The SM program is the implementation of the model defined by the experiment description. This description, made by the physicist in terms of the SM language (SML), is processed by the SMI tools in order to produce a State Manager and its associated proxies. So, the problem of creating a control system is therefore equivalent to that of giving a good description of the experiment in terms of the objects being controlled and the procedures that operate on them.

2.2 SMI

SMI is a tool for developing control systems, it is based on the concept of Finite State Machines (FSM). Finite state machines are a simple way to describe control systems and complex systems, this latter can be broken down into small and simple FSMs that are hierarchically controlled by other FSMs.

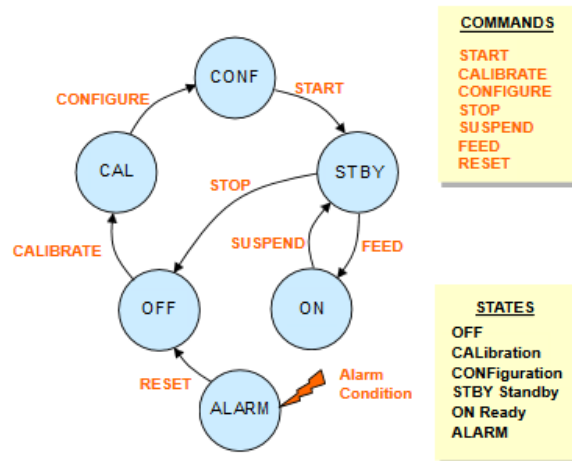


Figure 2.2 FSM example

Using SMI the experiment can be decomposed and described in terms of objects behaving as finite state machines. This model of interaction between objects is one that reflects our abstraction of reality.

2.2.1 Object model

SMI objects can represent directly a concrete entity in the experiment (such as “gas valve 12”) or it may equally represent a logical subsystem, or any abstraction used in describing the experiment provided it can be identified by a noun (e.g. “run”, “trigger”, “central detector”). The objects representing concrete entities interact with the hardware they model and control through driver processes or proxies. The main attribute of an object is its state, visible by other objects. The normal practice is to define values that correspond to adjectives which could qualify the name of the object (e.g. the object “run” may be in one of the states “active”, “dormant” or “paused”).

2.2.2 Object communication

An object operates on other objects by sending command messages to them. The communication mesh linking the objects carries the global flow of control in the system. A node can send state update messages to its parents and command messages to its children. The sources have no parents but can receive commands from other external control systems or from human operators.

A domain defines the scope of visibility for the associated objects’ names, it establishes the boundaries of the SM control space. This is achieved by assigning a name to each domain. The domain structure is superimposed in a completely transparent way.

2.3 Design the control system

The control problem is progressively narrowed down to a manageable level of complexity.

First of all, indicate the broad areas where an independent control activity is necessary, these areas will eventually become SM domains. For examples: “run control” or “detector control”, if the apparatus is composed of several detectors, each of them may belong to an independent SM domain. Then, for each SM domain, determine the controllable items, real entities that can be driven by commands and are susceptible of assuming a state; for each of them, a program will be provided in order to drive all the activity related to this entity, according to the instructions received by the SM.

A model of the behaviour of the given domain must be formulated using SML, as a SM program (Figure 2.3). The SM program describes the objects previously defined, the states they can take and the commands they accept in each state. The actions generated by each command must then be described: they consist of a sequence of instructions, each instruction being an executable statement of the experiment description language.

The service programs, or proxies, must be prepared as well. These programs operate on the apparatus upon receiving command messages. Their behaviour must be represented by a state machine.


```

1 class: $FWPART_$TOP$DcsCuType_CLASS
2 !panel: DcsCuType.pnl
3 state: NOT_READY !color: FwStateOKNotPhysics
4   when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
5   when ( $ALL$FwCHILDREN in_state READY ) move_to READY
6   action: CONFIGURE !visible: 1
7     do CONFIGURE $ALL$FwCHILDREN
8 state: READY !color: FwStateOKPhysics
9   when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
10  when ( $ANY$FwCHILDREN in_state NOT_READY ) move_to NOT_READY
11  action: RESET !visible: 1
12    do RESET $ALL$FwCHILDREN
13 state: ERROR !color: FwStateAttention3
14  when ( $ALL$FwCHILDREN not_in_state ERROR ) move_to NOT_READY
15  action: RECOVER !visible: 1
16    do RECOVER $ALL$FwCHILDREN
17

```

Figure 2.3 Simple example of State Manager Program

2.4 SML

Objects behave as finite-state machines.

A state in FSM describes the behaviour of a node when it is in that state, and consists of a, possibly empty, list of when clauses and a, possibly empty, list of action clauses. A change of state is brought about by the receipt of a command. Once accepted, a command triggers the execution of an action; this eventually terminates when the object reaches a new steady state. An action is identified by a verb applicable to the object name (e.g. the object “run” may perform any of the actions “start”, “pause”, “continue”, and “stop”).

An action consists of a sequence of operations, specified in the SM language by a list of instructions. These are essentially of two types: DO and IF. DO is an instruction that sends a command to another object; IF is an instruction that evaluates a Boolean function of the states of other objects, and makes a conditional branch depending on the result. While an object is executing an action, its state is undefined and no further command is accepted until the action is terminated. The final state reached after the execution of an action may depend on the results of tests performed on the state of other objects.

A command is not the only way of triggering an action: a state change in another object may provoke it. This type of dependence is specified in the SM language by a WHEN clause, this instruction are probed in the order in which they are given until one is found for which the guard is true.

See *Appendix A* for language keywords.

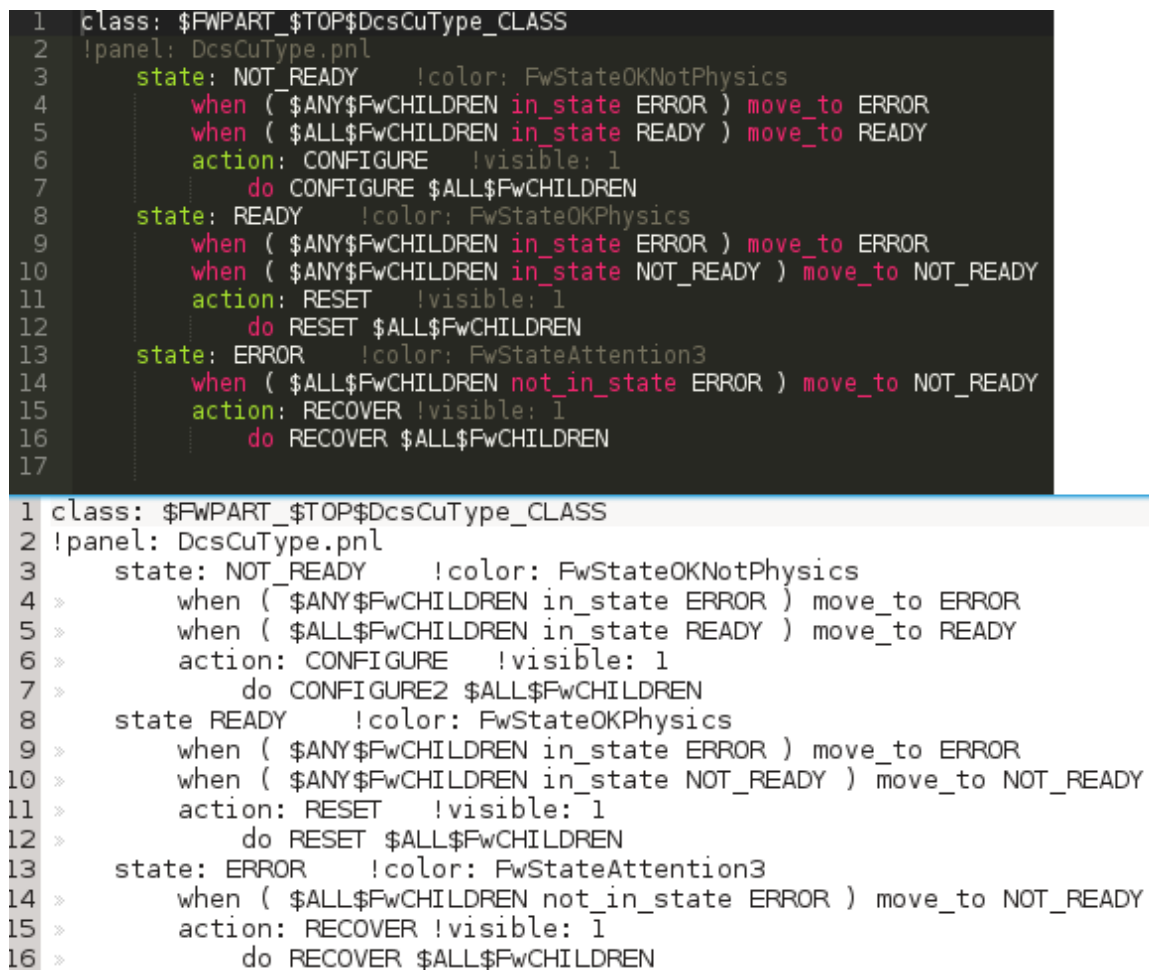
For this language an EBNF grammar [2] has been build, following rules in *Appendix B*.

3 Tools

3.1 ACE

Ace is an embeddable and standalone code editor written in JavaScript that performs syntax highlighting, automatic indent and outdent, searching and replacing of regular expressions, highlighting matching parentheses, dragging and dropping text using the mouse, code folding and many other.

The syntax highlighting allows code to be highlighted based on the language it's written in, to improve the readability and context of the text.



```

1 class: $FWPART_$TOP$DcsCuType_CLASS
2 !panel: DcsCuType.pnl
3 state: NOT_READY !color: FwStateOKNotPhysics
4 when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
5 when ( $ALL$FwCHILDREN in_state READY ) move_to READY
6 action: CONFIGURE !visible: 1
7 do CONFIGURE $ALL$FwCHILDREN
8 state: READY !color: FwStateOKPhysics
9 when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
10 when ( $ANY$FwCHILDREN in_state NOT_READY ) move_to NOT_READY
11 action: RESET !visible: 1
12 do RESET $ALL$FwCHILDREN
13 state: ERROR !color: FwStateAttention3
14 when ( $ALL$FwCHILDREN not_in_state ERROR ) move_to NOT_READY
15 action: RECOVER !visible: 1
16 do RECOVER $ALL$FwCHILDREN
17
1 class: $FWPART_$TOP$DcsCuType_CLASS
2 !panel: DcsCuType.pnl
3 state: NOT_READY !color: FwStateOKNotPhysics
4 > when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
5 > when ( $ALL$FwCHILDREN in_state READY ) move_to READY
6 > action: CONFIGURE !visible: 1
7 > do CONFIGURE2 $ALL$FwCHILDREN
8 state READY !color: FwStateOKPhysics
9 > when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
10 > when ( $ANY$FwCHILDREN in_state NOT_READY ) move_to NOT_READY
11 > action: RESET !visible: 1
12 > do RESET $ALL$FwCHILDREN
13 state: ERROR !color: FwStateAttention3
14 > when ( $ALL$FwCHILDREN not_in_state ERROR ) move_to NOT_READY
15 > action: RECOVER !visible: 1
16 > do RECOVER $ALL$FwCHILDREN

```

Figure 3.1 Comparison between highlight and raw input file

3.1.1 Set up environment

Ace can be easily embedded into a web page, downloading a prebuilt version from [ace-builds](#) repository, it contains 4 versions:

- src, concatenated but not minified
- src-min, concatenated and minified with uglify.js
- src-noconflict, uses ace.require instead of require
- src-min-noconflict.

Then creating a simple HTML web page.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>ACE in Action</title>
</head>
<body>

<div id="editor"> TEXT TO BE HIGHLIGHTED </div>

  //path to ace mode
<script src="ace-builds-master/src-noconflict/ace.js"
type="text/javascript" charset="utf-8"></script>
<script>
  var editor = ace.edit("editor");
  editor.setTheme("ace/theme/xcode");
  editor.getSession().setMode("ace/mode/fsm");
</script>
</body>
</html>
```

Finally, create a mode, a JavaScript file that points to definitions for the highlighting rules, as well as rules for code folding. Without defining a mode, ACE won't know anything about the finer aspects of the language.

```
// defines the language specific highlighters and folding rules
var MyNewHighlightRules =
require("../mynew_highlight_rules").MyNewHighlightRules;
var MyNewFoldMode = require("../folding/mynew").MyNewFoldMode;

var Mode = function() {
  // set everything up
  this.HighlightRules = MyNewHighlightRules;
  this.$outdent = new MatchingBraceOutdent();
  this.foldingRules = new MyNewFoldMode();
};
oop.inherits(Mode, TextMode);
```

3.1.2 ACE's rules

The Ace highlighter can be considered to be a state machine, in which regular expressions define the tokens for the current state, as well as the transitions into another state. The token state machine operates on whatever is defined in this.\$rules. The highlighter always begins at the start state, and progresses down the list, looking for a matching regex.

```
var MyNewHighlightRules = function() {

  // regexps are ordered -> the first match is used
```

```

this.$rules = {
  "start" : [
    {
      token: <token>, // String, Array, or Function: the CSS
token to apply
      regex: <regex>, // String or RegExp: the regexp to
match
      next: <next> // [Optional] String: next state to
enter
    }
  ]
};
};

```

Let introduce a simple example to understand the roles of token, regex and next. The syntax highlighting state machine stays in the `start` state, until you define a `next` state for it to advance to. At that point, the tokenizer stays in that new state, until it advances to another state. Afterwards, you should return to the original `start` state.

```

this.$rules = {
  "start" : [ {
    token : "text",
    regex : "<\\!\\\[CDATA\\\[",
    next : "cdata"
  },
  "cdata" : [ {
    token : "text",
    regex : "\\]\\]\\]>",
    next : "start"
  }, {
    defaultToken : "text"
  } ]
};

```

When `<![CDATA` tag is encountered, the tokenizer moves from `start` into the `cdata` state. It remains there, applying the `text` token to any string it encounters. Finally, when it hits a closing `]>` symbol, it returns to the `start` state and continues to tokenize anything else.

The Ace highlighting system is derived by TextMate [5] language grammar. A language grammar is used to assign names to document elements such as keywords, comments, strings or similar to allow styling. In ACE editor this role is carry out by the `token` identifier, which represent the type of the regular expression defined in `regex`. For example:

```

{
  token : "constant.language.boolean",
  regex : /(?:true|false)\b/
}

```

3.2 PEG.js

PEG.js is a simple parser generator for JavaScript, the name is the acronym for Parsing Expression Grammar. Parsing is the general problem of turning raw text into structured data. In programming language theory grammar is considered a context free grammar (CFG), and in particular, parsing converts the text the programmer writes into an abstract syntax tree.

A CFG is a set of production rules involving non-terminal and terminal symbols. Non-terminal symbols are things like expression or integer, they are names that help parse input but aren't actually part of the input.

```
S -> " "
```

```
S -> "( " S " ) "
```

```
S -> "[ " S " ] "
```

The CFG represented in the example above accepts a period with any number of matched parentheses and square brackets.

In PEG there is no need to declare precedence levels and associativity, because if PEG finds more than one way to match an expression, the first match is chosen. This rule is chosen in order to avoid ambiguity [10].

The most important restriction in a PEG is that it is not possible to have left recursive rules because it leads to an infinite loop in the parser. For example, the rule $E \rightarrow E "+" E$ is left recursive because the non-terminal symbol appears in the first position of the body of the rule.

3.2.1 What is a parser

A parser is a program, usually part of a compiler, that receives input in form of sequential source program instructions, interactive online commands or markup tags and breaks them up into parts that can then be managed by other programming.

Parsers are used to recognize "structure" of language phrases, such structure is generally "context sensitive", and that means that parsers attach meaning by classifying strings of tokens from the input (sentences) as the particular nonterminals and building the parse tree. E.g.: all these token strings: `[number][operator][number],[id][operator][id],[id][operator][number][operator][number]` will be classified as "expression" nonterminal by the C/C++ parser.

Must be remarked that the concept of a lexer is different by the parser concept. Lexers are used to recognize "words" that make up language elements. While lexers are about getting the words right, parsers are about getting the sentences right. "See spot run" and "spot run See" are both valid as far as a lexer is concerned, but it takes a parser to determine that phrase structure is wrong.

3.2.2 Set up environment

PEG.js can be installed through Node.js or from the browser.

In order to install PEG.js with Node.js [7]:

```
install PEG.js globally $ npm install -g pegjs
```

```
install PEG.js locally  $ npm install pegjs
```

In order to install PEG.js with the browser, download the PEG.js library (regular or minified version) or install it using Bower [8]

```
$ bower install pegjs
```

3.2.3 Generate the parser

PEG.js generates parser from a grammar that describes expected input. A parser can be generated in three ways:

- From command line:

```
$ pegjs [OPTIONS] grammar.pegjs grammar-parser.js
```

By default, the parser object is assigned to `module.exports`, which makes the output a Node.js module. You can assign it to another variable by passing a variable name using the `-e/--export-var` option. This may be helpful to use the parser in browser environment.

3.2.4 PEG.js's rules

Starting from a grammar, PEG.js creates a parsing function. This function `parse` takes a string as input.

A grammar for PEG.js is a list of rules, based on RegEx [9].

Each rule has a name (`sum - digit`) that identifies the rule and then a body (ex: `[0-9]`) that defines a pattern to match and can contains some JavaScript code that determines what happen if the pattern matches successfully (`{return parseInt("" + first + second + third);}`). The start name is special because parsing always starts there.

The first rule can be preceded by an initializer, a JavaScript code, which is executed before the parser starts parsing.

Grammar example:

```
{
  Initializer
}

start =
  sum

sum =
```

```

    first:[1-3] second: digit third: digit

    {return parseInt("" + first + second + third);}

digit =

    [0-9]

```

If an expression matches a part of the input text when running the parser, it produces a matched result. The match results propagate over the rules, up to the `start` rule. Then the parser returns start rule's match result when parsing is successful.

Input	Output
123	6
a	throw an exception indicating that the input could not be parsed.

4 Development and Implementation

4.1 ACE

ACE is a code editor written in JavaScript (see section 3.1), that matches the feature of native code editors such as Sublime and Vim, and offers real time checking for code accuracy. It provides syntax highlighting, proper indentation, keyboard shortcuts, auto-completion, code folding, find and replace of regular expressions.

4.1.1 Mode Rules

In this paragraph is given an outline of the construction of the highlighter and the chosen rules.

4.1.1.1 Define a html page to render the input text:

```

<!DOCTYPE html>
<html lang="en">
  <head>
    . . .
  </head>
  <body>

    <div id="editor">
      Input text
    </div>

```

```

<script src="ace-builds-master/src-noconflict/ace.js"
type="text/javascript" charset="utf-8"></script>
<script>
    var editor = ace.edit("editor");
    editor.setTheme("ace/theme/xcode");
    editor.getSession().setMode("ace/mode/fsm");
</script>
</body>
</html>

```

`editor.getSession().setMode("ace/mode/fsm");` defines the path to the mode that contains the syntax highlighting rules, indentation rules, and code folding rules.

4.1.1.2 Define the ace mode rules

The highlighting rules have been created for the CU Type, following the definition of the SML in the document “*Validation of CERN's Finite State Machines*” [2].

- Comments are allowed in the input text in the form `!comment[EOL]`. A single line comment must have the specific type `comment.line.*`, a comment line can be of many types (`.*`) depending of the starting character (`#` for example).

```

{ token: 'comment.line.*',
  regex: '//.*$|!.*$' },

```

- The syntax has two main entities (`state` and `action`) that must be checked during the parsing in order to have coherent relations between states and between actions on states (more lately in the PEG section). In order to identify them properly, they are defined with a predefined colouring. The entity token has been defined because an entity refers to a larger part of the document, for example a chapter, class, function, or tag; as well as, in SML, it referred to an “entity” whose role is to define a “container” for the node.

```

{ token: 'entity.name.function',
  regex: '\\b(?:state)\\b' },

```



Figure 4.1 state and comment highlighting

```

{ token: 'entity.name.function',
  regex: '\\b(?:action)\\b' }

```

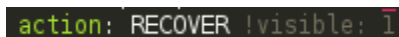


Figure 4.2 action and comment highlighting

- The SML has also keyword related to the control flow of actions

```

{ token: 'keyword.control',

```



```

regex:
  '\\b(?:when|do|set|wait|sleep|if|then|else|endif)\\b'
},
{ token: 'keyword.operator',
  regex: '\\b(?:or|and)\\b' }

```

and keyword related to specific operations/controls that can be performed on the state

```

{ token: 'keyword.operator',
  regex:
  '\\b(?:move_to|in_state|not_in_state|stay_in_state)\\b
' }

```

```
when ( $ALL$FwCHILDREN not_in_state ERROR ) move_to NOT_READY
```

Figure 4.3 control keyword and operator

- Another important aspect that has been analysed was the folding concept. Folding has been defined for round and square parenthesis and curly brackets.

```

this.foldingStartMarker=/(\{|\(|\[|^}\}\|)]*$|^\\s*(\\/\\*)/;
this.foldingStopMarker=/^[^\\{\\}]*\\(|\\)||^\\s\\s*(\\*\\/\\*)/;

```

An interesting chance would have been done folding based on `state` or `class`, in order to be able to aggregate all the information about a state. In order to support code folding, the text editor must provide a mechanism for identifying "folding points" within a text file, a beginning and an end. The problem in providing this feature has been finding an end point for the folded block: the language define a start point, `start:`, but does not provide any final keyword or parenthesis, neither was possible to use the same word as an end. Tabulation was not possible because is allowed at every start of a line.

4.2 PEG

As described in section 3.2 PEG.js is a parser used to evaluate syntax rules of a grammar, SML in case of Control Unit Type. The FSM syntax rules are described in Appendix C and Chapter 3 of *Validation of CERN's Finite State Machines* [2].

4.2.1 Creation of Syntax Rules On-line

PEG.js tool offers an online version of the syntax parser [13], which has been used for the generation of the `.pejgs` grammar, from which the parser is finally generated.

In the following is listed a sequence of rules that have been implemented:

- Syntax rules
 - Every rules of Appendix C [2] has been implemented. Here an example to show how the RegEx work

```

FSM syntax
  identifier = character, {character};

```

```
action clause = 'action:', [action parameter],
identifier, statement, {statement};
```

RegEx syntax

```
action =
    "action:" + space*
identifier =
    [0-9a-zA-Z'_-'&']
statements =
    first:statement enter*
action_parameter =
    "(" space* third:action_parameter_list space*
    ")"
action_simple =
    action enter* second:action_parameter? enter*
    third:identifier+ comment* fourth:statements+
```

In the following table, are explained some relations that are between the FSM syntax and the RegEx rules [9]:

FSM syntax	RegEx syntax
[action parameter] [] indicate that the token inside is optional, so it can be taken 0 or 1 time.	action_parameter? The correlation between the [] and the RegEx is made thanks to the '?' character, that indicates that the non-terminal parameter can be taken/not-taken
identifier = character, {character}; + indicates that 1 or more token can be taken from the terminal parameters that derive from the non-terminal <u>identifier</u>	identifier+ + indicates that 1 or more token can be taken from the terminal parameters that derive from the non-terminal <u>identifier</u>

- One fundamental point for the project has been the comparison between tokens that represent states and actions in the regular expression. For this purpose, JavaScript functions have been used. Let's take as an example the code below to define in a clear way the problem.

```
state: ERROR !color: FwStateAttention3
when ( $ALL$FwCHILDREN not_in_state ERROR ) move_to NOT_READY
action: RECOVER !visible: 1
do RECOVER $ALL$FwCHILDREN
```

- Every state has a name (ERROR), which identify the node in the FSM. Every state can have a sequence of 'when sentences' (second line) and 'action sentences' (third and fourth line); a when sentence is defined by an expression (`ALLFwCHILDREN not_in_state ERROR`) and by a referrer (`move_to NOT_READY`). An action sentence is identified by a name (RECOVER), and can have a sequence of statements (`do RECOVER ALLFwCHILDREN`) that represent the action that must be performed on other states (`ALLFwCHILDREN`).
 - State checking: states' names must be recorded in order to not perform actions on states that do not exist. In the previous example, the `move_to` action cannot be performed on state `NOT_READY` because it does not exist. So an error must be thrown by the parser.
 - State child checking: the same checking defined before must be performed with respect to the state declared in the children's nodes (`ALLFwCHILDREN`). In the previous example, if the state `ERROR` does not exist in all the children nodes of the class in which the when expression is defined (`ALLFwCHILDREN`) an error is thrown by the parser.
 - Action checking: the same check done for the states must be performed between actions. An action can be performed on a state only if it is declared.

4.2.2 Generation of the Parser

After the definition of the grammar, there are two ways to create a parser:

- From the on-line version [13]: define a parser name before downloading it (ex: parser)
- From the shell: save the grammar file as *grammar.pegjs* and execute the command


```
pegjs [OPTIONS] grammar.pegjs grammar-parser.js
```

 With the optional parameter `--trace`, when the input file is parsed, the output file shows also the sequence of used RegEx rules.

4.2.3 Input File Checking

- From html page: the content of the input file needs to be copied into a textarea, to be processed by the parser.
- From shell: to run the parser execute the command `node parser.js`
The input file must be read through `parser.js`

4.2.4 Error reporting

The input text is parsed in a sequential order. As soon as the parsers detects an error, the process is stopped and the error is reported.

5 Future Scope

5.1 Integration with WinCCOA

Coworkers in the section have already started to integrate the CU editor with the WinCCOA [11] panel.

WinCC-OA, previously known as PVSS, is a SCADA framework owned by Siemens, which support the control system processes at CERN. For example, applications for monitoring and controlling detectors of major experiments, ventilation, cryogenics, cooling systems and other infrastructures.

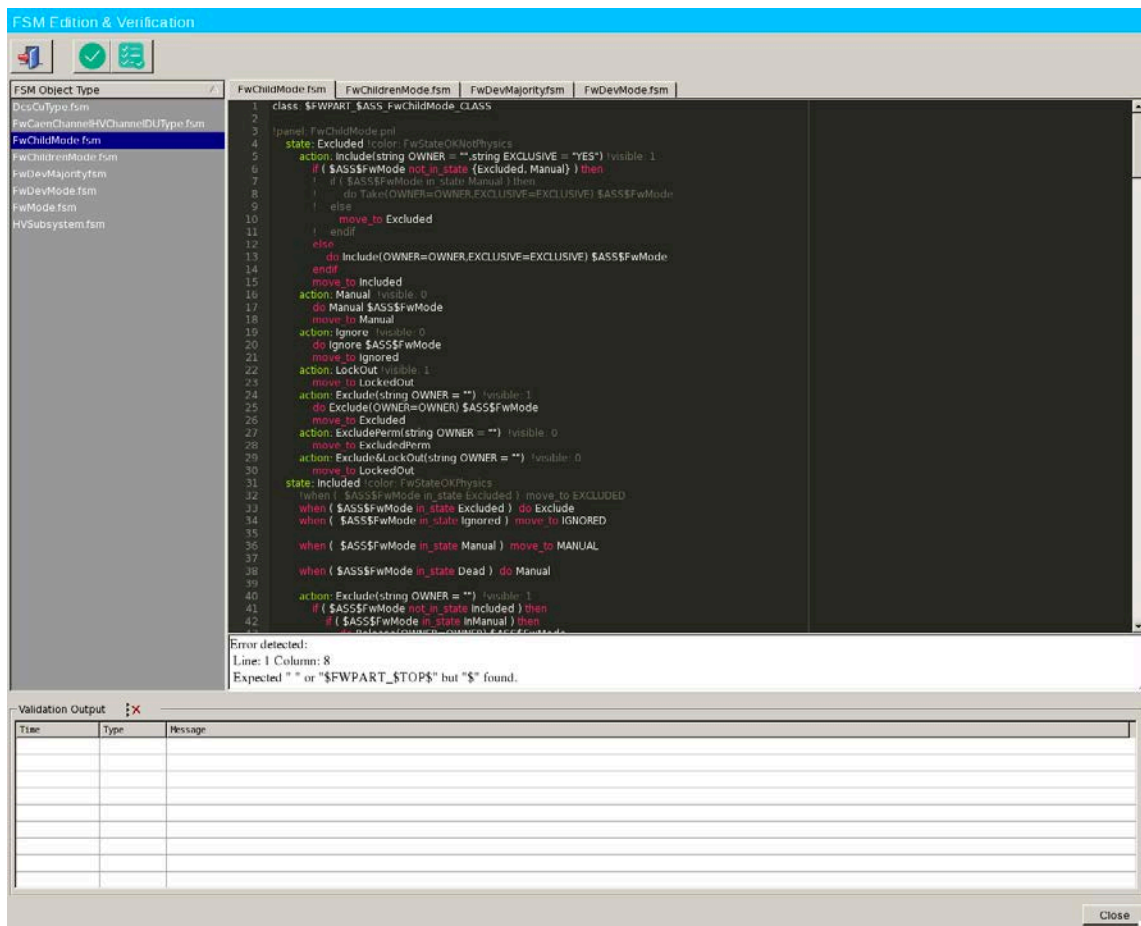


Figure 5.1 WinCCOA integration

5.2 Create Final State Machine Visualization

Another possible usage of this tool is the ability to build a FSM starting from the input file. This approach can be extremely useful to have a graphical vision of the system.

For example from this simple input file

```
class: $FWPART_STOP$DCUtypeClass
```

```

state:NOT_READY

    when ($ALL$CHILDREN in_state NOT_READY) move_to ERROR

    when ($ALL$CHILDREN in_state READY) move_to ready

state:READY

state:ERROR

```

This kind of FSM can be obtained for each class.

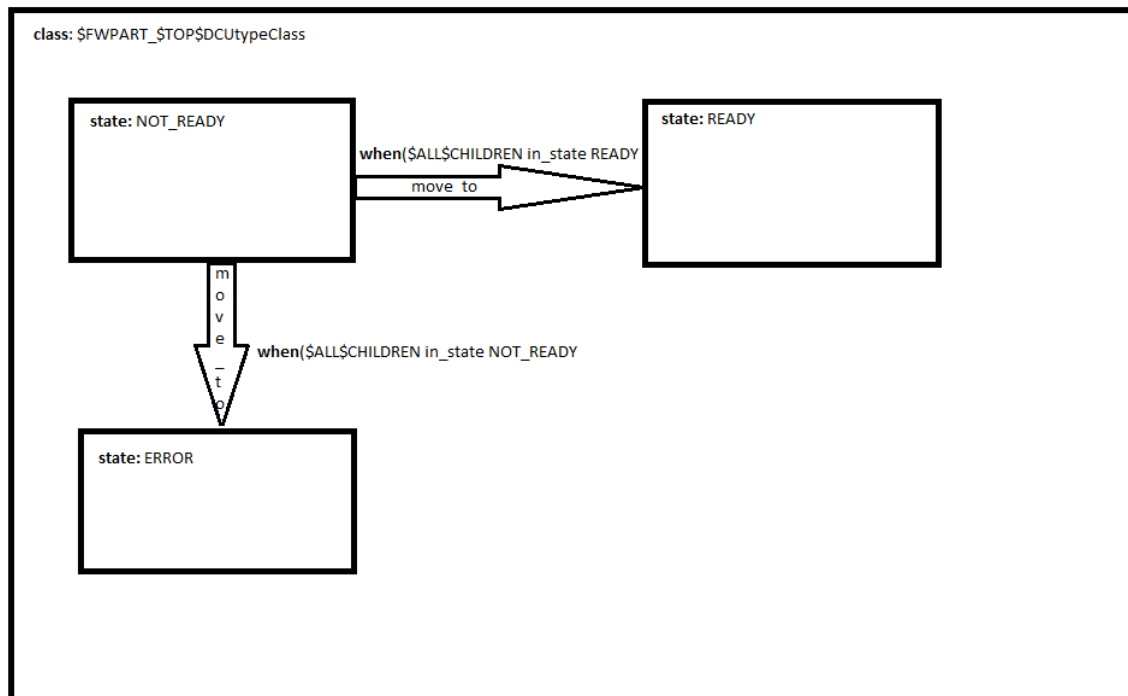


Figure 5.2 FSM example

Each class diagram must be connected to its parents' and to its childs' FSMs, in order to do so, another graphical representation can connect classes to classes.

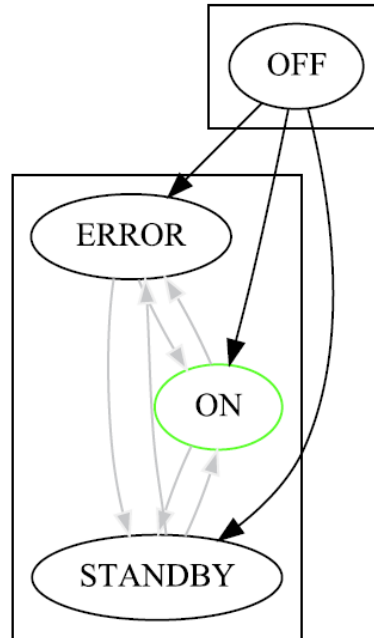


Figure 5.3 FSM between classes

A possible tool to create FSMs is Concrete Editor [12].

6 Conclusions

This tool proposes a useful approach to facilitate the **visualization and error checking** of the of FSM Control Unit Type online. It gives a visually comprehensive method to simplify code checking thanks to the syntax highlighter, and ensure a correct evaluation of the rules of the code, as well as a coherent relation between states, thanks to the parser.

To prove the relevance of the tool, the integration with WinCCOA panel must be completed. And possible improvements can be done thanks to future proposals.

Appendix A

A formal definition of the language syntax can be found in [1/Appendix A] and [3].

CLASS declaration

```
CLASS: class_name [/ASSOCIATED]
```

This statement is used to describe a set of identical objects. The description of the set begins with the CLASS declaration. All the statements concerning the set of objects must immediately follow. For the ASSOCIATED class, the same considerations apply as for the associated object (see the OBJECT declaration).

OBJECT declaration

```
OBJECT object_name [/ASSOCIATED] [IS_OF_CLASS class_name]
```

The description of an object begins with the OBJECT declaration. All the statements concerning one object must follow immediately its declaration.

An ASSOCIATED object does not execute itself the action it receives. It transmits the command to its associated process and updates its state according to the variation of the associated process. The associated process may belong to a domain different from the one of the SM itself: in this case the domain must be explicitly specified as a prefix of the object name, separated from it by a double colon (e.g. TPC::HV indicates the object HV in the domain TPC).

The IS_OF_CLASS qualifier specifies that the object inherits the description and all the characteristics of the class.

Both objects (abstract and associated) can have parameters associated with it. One of their most important uses is passing values between SMI world and the associated proxies. When object has parameters, then the Object declaration must be immediately followed by:

```
Parameters : parameters-declaration

parameters-declaration : pd [, pd,...,pd]

pd : [type] name [ = default-value]
```

STATE declaration

```
STATE : state_name [/INITIAL_STATE] [/DEAD-STATE]
```

This statement is used to declare one of the possible states an object can take. The STATE declaration may be followed by all the WHEN and ACTION statements that apply to this state.

INITIAL_STATE, in the case of logical objects, indicates that the object must be put in the flagged state when the SM program starts up, in the case of associated objects, the flagged state is the one that the object initially assumes when a process becomes associated with it. By default, the initial state is the first one declared.

DEAD_STATE is a state qualifier which only applies to associated objects. It designates the state to be assigned to an object when the associated process aborts or in general is not running.

ACTION declaration

```
ACTION : action_name [(param [= default_value], ...)]
```

This statement declares a command accepted by the object in a given state. If the object accepts the same command in different states, this command must be declared for all the states.

For a logical object the ACTION declaration must be followed by the instructions that the object performs at reception of the command. On the other hand, an associated object does not execute any action but sends the command as a message string to the associated process.

WHEN statement

```
WHEN condition DO action_name
```

The WHEN statement allows an object to react to unsolicited state changes of some other objects. The statement applies to one state only: if the logical condition becomes true when the object is in this state, the object starts to execute the action specified in the WHEN statement.

The condition of the WHEN statement can be a logical expression combining the state of various objects with logical operators.

DO statement

```
DO action_name [(parameter=value,...)] object_name
```

This statement allows an object to trigger further actions on itself or on other objects during the execution of an action.

IF statement

```
IF condition THEN statements [ELSE statements] ENDIF
```

The IF statement conditionally executes a statement or a block of statements. The condition can be a logical expression combining the state of various objects with logical operators.

The IF statement realises also the synchronisation between all the objects whose state is tested in the condition. Before evaluating the condition, all the objects will execute the pending actions.

Appendix B

This appendix gives the Extended Backus-Naur Form (EBNF) grammar of the FSM syntax. White space and comments are left out of this grammar but are allowed at every comma (,) in the grammar, except if indicated otherwise. Comments in FSM are embraced with an exclamation mark (!) and a line end. The start symbol is `specification`.

```
specification = class, {class};
```

```
class = 'class:', '$FWPART_$TOP$', identifier, state, {state};
```

```
state = 'state:', identifier, {when clause}, {action clause};
```

```
when clause = 'when', expression, referrer;
```



```
expression = (paren expression | not expression | base expression),
              [and expression | or expression];
base expression = child pattern, ( 'empty' |( state operator,
              state specification ) );
state operator = 'in_state' | 'not_in_state';
not expression = 'not', '(', expression, ')';
paren expression = '(', expression, ')';
and expression = 'and', expression;
or expression = 'or', expression;
child pattern = '$', ['all$' | 'any$'], identifier;
state specification = identifier |
                    ('{', {identifier, ','}, identifier, '}');
referrer = referrer do | referrer move_to | referrer stay_in_state;
referrer do = 'do', identifier;
referrer move_to = 'move_to', identifier;
referrer stay_in_state = 'stay_in_state', [identifier];
action clause = 'action:', [action parameter], identifier,
                statement, {statement};
action parameter = '(', [ action parameter single, {'',
                action parameter single} ], ')';
action parameter single = 'string', identifier,
                          '=', string literal;
string literal = '"', {? any character but " ?}, '"';
statement = statement do | statement move_to |statement if |
            statement set | statement wait | statement sleep;
statement do = 'do', identifier, ['(', statement parameter, ')'],
```

```
child pattern;

statement parameter = ['string'], identifier, '=', (string literal
    | identifier | statement parameter object);
statement parameter object = '$', identifier, {'.', identifier};
statement move_to = 'move_to', identifier;
statement if = 'if', expression, 'then', statement, {statement},
    ['else', statement, {statement}], 'endif';
statement set = 'set', statement parameter;
statement wait = 'wait', '(', child pattern, {',',
    child pattern}, ')';
statement sleep = 'sleep', integer;
identifier = character, {character};
(*in an identifier, no white space or comment is allowed*)
(*after an identifier, no 'character' is allowed*)
integer = digit, {digit | '0'};
(*in an integer, no white space or comment is allowed*)
character = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
    'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
    'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
    'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' |
    'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' |
    'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' |
    'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' |
    'X' | 'Y' | 'Z' | '_' | '&' | '-' | '0' |
digit;

digit = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
```

References

- [1] *SMI++ User Manual*, B.Franek, C. Gaspar, 28 March 1988
- [2] *Validation of CERN's Finite State Machines*, Sander J.J. Leemans, 8th June 2012
- [3] http://smi.web.cern.ch/smi/LANGUAGE_DESCRIPTION.html
- [4] <https://ace.c9.io/#nav=about>
- [5] http://manual.macromates.com/en/language_grammars
- [6] <http://pegjs.org/>
- [7] <https://howtonode.org/how-to-install-nodejs>
- [8] <http://blog.teamtreehouse.com/getting-started-bower>
- [9] http://www.w3schools.com/jsref/jsref_obj_regexp.asp
- [10] <http://books.xmlschemata.org/relaxng/relax-CHP-16-SECT-1.html>
- [11] <https://wikis.web.cern.ch/wikis/display/EN/PVSS>
- [12] <http://concrete-editor.org/>
- [13] <http://pegjs.org/online>