FINITE STATE MACHINE PREPROCESSOR FOR PVSS SCADA SYSTEM AT TOTEM EXPERIMENT

Bachelor's Thesis

Sami Stöckell

Degree Programme in Information Technology
Information system management

Accepted ___.___._____ _____

SAVONIA-AMMATTIKORKEAKOULU TEKNIIKKA KUOPIO

Koulutusohjelma

Tietotekniikan Koulutusohjelma

Tekijä

Sami Stöckell

Työn nimi

Äärellisen Tilakoneen Esikäsittelijä TOTEM Kokeen PVSS SCADA Järjestelmälle

Työn laji	Päiväys	Sivumäärä
Insinöörityö	26.04.2010	47 + 1

Työn valvoja	Yrityksen yhdyshenkilö
Lehtori Keijo Kuosmanen	Fernando Lucas Rodriguez

Yritys

CERN TOTEM koe

Tiivistelmä

Tämän opinnäytteen tavoitteena oli suunnitella ja ohjelmoida esikäsittelijä PVSS ohjelmistolle. Esikäsittelijän avulla voidaan suunnitella ja automatisoida hajautettuja ohjausjärjestelmiä. Esikäsittelijä on suunniteltu helpottamaan ja nopeuttamaan ohjausjärjestelmän rakentamista ja päivittämistä automatisoinnilla. Esikäsittelijän avulla käyttäjä voi asentaa ja päivittää ohjausjärjestelmän kuten ohjelmistopaketin. Tämän opinnäytteen tilaajana toimi TOTEM koe CERNissä.

Esikäsittelijän nimeksi annettiin FSMtool ja se suunniteltiin käyttäen C Sharp 3.0 ja .NET ohjelmointikieltä. FSMtool käyttää sisäänrakennettua heuristiikkaa luodakseen loogiset ja laitteistonimet sekä äärellisen tilakoneen. Yhdessä nimet kuvaavat PVSS järjestelmän laitteisto- sekä ohjelmistokomponenttien välisen yhteyden ja äärellinen tilakone kuvaa ohjausjärjestelmän logiikan. Säännöt joilla ohjelma luo nämä tiedot ovat ohjelmoitu käyttäen XML kieltä sekä säännöllisiä lausekkeita.

Ennen FSMtoolia nämä yhteydet syötettiin ohjausjärjestelmään käsin, nyt automatisoidusti. FSMtool oli onnistunut projekti ja sen avulla työmäärä väheni sekä järjestelmän päivittäminen nopeutui. Projektin suurimmat haasteet olivat suunnitella ohjelman säännöt helposti päivitettäväksi ja ohjelmoida äärellisen tilakoneen kuvaava rakenne. Eräs FSMtoolin vaatimuksista oli käyttää Microsoft Office Excel tiedostoja ohjausjärjestelmän kuvaavina tiedostoina.

Avainsanat

CERN, TOTEM, PVSS.

Luottamuksellisuus

julkinen

SAVONIA UNIVERSITY OF APPLIED SCIENCES

Degree Programme

Information Technology

Author

Sami Stöckell

Title of Project

Finite State Machine Preprocessor for PVSS SCADA System at TOTEM Experiment

Type of Project	Date	Pages
Final Project	April 26, 2010	47+1

Academic Supervisor	Company Supervisor
Mr. Keijo Kuosmanen, Principal Lecturer	Mr. Fernando Lucas Rodriquez.

Company

CERN TOTEM Experiment

Abstract

The aim of this final project was to create a preprocessing software for the PVSS supervisory control and data acquisition system to make the update process of the system faster. With PVSS the user is able to design and build large scale scattered control systems. The preprocessing software was designed to make updating and commissioning of PVSS systems faster and easier with automation. This thesis was commissioned by Total Cross Section, Elastic Scattering and Diffraction Dissociation experiment at CERN.

The preprocessor was named as the FSMtool and developed using C Sharp 3.0 and .NET coding languages. The FSMtool uses built-in heuristics to generate logical names, hardware names and finite state machine hierarchy for the operation logic of the control system. These names and hierarchy are needed when connecting control system software with hardware in PVSS. The control system heuristics that describe creating this logic and connectivity are programmed using XML and use regular expressions as an identification method when the rules are applied to the data.

The FSMtool was successfully implemented and it reduced the time and workload required to build the system. The biggest challenges of the project were to build the program to be easily updated and the programming of the finite state machine hierarchy structure. One criterion for the project was to have the people use the Microsoft Office Excel file format to store the connectivity data of the control system.

Keywords

CERN, TOTEM, PVSS.

Confidentiality

public

Acknowledgements

I thank Fernando Lucas Rodriguez for his invaluable help and guidance during the developing and writing of the code for the FMSTool. I also want to thank Keijo Kuosmanen for supervising the writing of this thesis. Arto Toppinen from Savonia University of Applied Sciences and Rauno Lauhakangas from Helsinki Institute of Physics for arranging the opportunity to write my final year project for the TOTEM experiment.

It has been a pleasure working with you.

April 26, 2010.

_____

Sami Stöckell

1 INTRODUCTION

CERN has six experiments in total and all of them use the PVSS II 3.8. as the program for building scattered control systems. Total Cross Section, Elastic Scattering and Diffraction Dissociation at the LHC, (TOTEM) has a detector control system group, known as THE DCS group. This group is responsible for designing, constructing, testing and maintaining the TOTEM experiment control system software. The idea behind the project is to be able to maintain large scale control systems at the experiment with an increased ease of use and to decrease the possibility of human made error in the creation of the control system.

The scattered control systems used in the experiments are built with the PVSS II 3.8 by clicking and dragging components to the graphical editor and entering all the data manually for the different components. This is time requiring, arduous and error prone work when larger systems are in question. The goal of this thesis is to be able to maintain an up to date detector control system with ease and design and build a preprocessing tool for automating the creation of the control system connectivity. The preprocessor is named as the FSMtool.

The purpose of the FSMtool is to read control system connectivity data from files and apply rules to the data for creating logical connectivity between hardware and software and creating a finite state machine that describes the logical functionality of the control system. To keep the FSMtool as maintainable as possible, the inner logic and the heuristics are described in XML. This helps the users by giving a possibility to update the information of the system to the FSMtool without prior knowledge of programming. All that is required for the use of the FSMtool is knowledge of XML structures and regular expressions. The FSMtool is designed to work either as a background process that receives commands of the PVSS from the command prompt and through a graphical user interface.

2 TOTEM

With roman pots positioned at distances of 147 meters and 220 meters from the Compact Muon Solenoid (CMS) interaction point and others inside CMS, the Total Elastic and Diffractive Cross Section Measurement, abbreviated as TOTEM experiment, will measure the total interaction cross-section of protons at the LHC. Figure 2.1 shows the main location of the TOTEM experiment. [2]

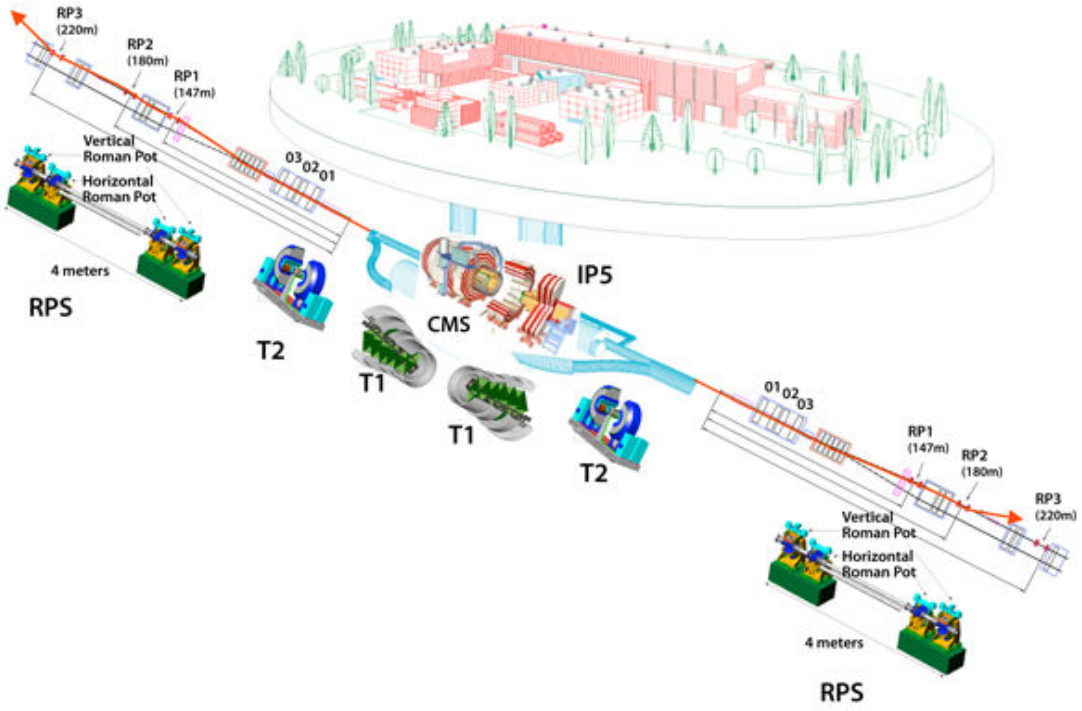


Figure 2.1. Main location of the TOTEM experiment. [1].

2.1 Experiment

The data collected by the experiment will help to improve knowledge of the internal structure of the proton and the principles that determine the shape and form of protons as a function of their energy. Specific to the TOTEM experiment are the Roman pots, these cylindrical vessels can be moved to within 1 mm of the beam centre. They contain detectors that will measure very forward protons which arise from elastic scattering and diffractive processes, only a few micro radians away from the beams.

Inelastic interactions between protons will be studied by gas electron multipliers, abbreviated as GEM, installed in so called telescopes, placed in the forward region of the CMS detector. Each of the telescopes contains 20 half-circle detectors arranged in 10 planes, with an inner radius matching the beam pipe. [2]

The closer the Roman pot detectors can get to the path of the beam, the more precise the results. For the LHC, the Roman pots will collect data from a distance of 800 μm from the beam. In the final configuration, eight Roman pots will be placed in pairs at four locations at Point 5 on the LHC. There are two stations at each end of the CMS detector, positioned at distances of 147 m and 220 m from the collision point, interaction point 5. TOTEM has now installed all the Roman pots and has equipped a few of them with detectors. This will allow testing the movement of the Roman pots with respect to the beams at the LHC start-up and to take some first data. The GEM detectors were also installed within CMS. [2]

2.2 Detector Control System

The Detector Control System, abbreviated as DCS, for LHC experiments are the evolution of slow control systems from the large electron-positron collider era from 1989 to 2000. Their aim is to permit the user on shift to operate and control various detector subsystems such as high and low voltage power supplies, gas circulation systems, cooling and so on and monitor their performance as well as various relevant environmental parameters. For example temperature and radiation levels. All DCS systems of LHC experiments are built using the industrial the PVSS control supervisory software running on networks of personal computers (Windows and/or Linux), augmented with modules developed at CERN for typical HEP control functions and equipment, the JCOP framework. The TOTEM DCS system also uses the same guidelines, technology, tools and components. [2]

The TOTEM detector is located at the CMS intersection of the CERN LHC accelerator. CMS and TOTEM also have a common physics program, so that TOTEM must be able to operate independently, but also take data together with CMS. For this reason TOTEM adopted a number of technological and organizational solutions that will permit interoperability at the level of electronics, data acquisition, run control,

offline data processing and so on. In the same way, the TOTEM detector control system is developed in the framework of the CMS DCS. [2]

The three TOTEM subdetectors, Roman Pot silicon detectors, T1 and T2, require the monitoring and control of the usual equipment found around particle physics experiments. All three use CAEN high voltage power supplies, Wiener Maraton low voltage units, Wiener VME crates, and environmental sensors connected to ELMBs or read from the DAQ through the DCU technology. The front-end is based on National Instruments hardware and the Labview RealTime software, adapted to the selected sensors. [2]

The system consists of hardware and software components, therefore it requires comprehensive system engineering. A number of technologies have been defined already before the project started: sensors, front-end systems, the supervisory level based on the PVSS and the JCOP Framework. Finite States Machines, abbreviated FSM, are used in describing the state of the system. [2]

3 PROBLEM DESCRIPTION AND REQUIREMENTS

This chapter explains the problems in PVSS administration that the FSMtool is designed to solve and the requirements for the FSMtool.

The control system is built on logical names and hardware names that create the connectivity between hardware and software of the PVSS. The PVSS uses a logical model based on finite state machine. Before The FSMtool, all of these names and models have been entered manually to the control system. The PVSS is based on the graphical user interface for creating control systems, meaning that the user needs to click as many as six to seven times to create one control component for the system. This means going through menus and submenus to find the right item. Then the item is dragged to the PVSS design board and all the properties, names and aliases must be entered manually.

The FSMtool is designed to remove this stage in the process of building the control system, by implementing the FSMtool to the PVSS, the time required to do this stage is reduced significantly. What previously could take days, even a week is now done in hours with the help of the FSMtool. The PVSS in itself is quite massive software and sometimes slow in processing speed when building the system. All of these factors add up to the time required to spend building the system. Manually done when there are hundreds of components, checking from the XLS table that contains the connectivity information, creating the component in the PVSS and checking that the information is correct takes a long time. Building large scale systems means a lot of work, from ground up to the phase where it is working can take a week from one person, when all the information is correct and available in the connectivity describing XLS tables. This time could be used more efficiently.

There is always a possibility for human error when building the system manually, as one needs to check data from documentation, paper or digital, to create one data point component with the appropriate information. Going through hundreds of lines of information stored in an XLS table where the information can be on tens of columns is manually done quite a feat. The FSMtool uses built in heuristics to write the data for the system, so if the heuristics are correctly entered, the errors made by the user are

reduced. If the user has entered some value incorrectly when entering data manually, it needs to be corrected and there is only one way to solve the problem. The data point has to be deleted and a new data point has to be created again manually. And if the whole section is entered incorrectly, it has to be done again. So yes, the PVSS is a system that is not so user friendly. This is a major flaw in the system, or as the PVSS says, a feature.

The unforgiving nature of these steps may require the user to double check to make sure no errors were entered and the data point is correctly built. When applied to large scale systems such as TOTEM, where there are hundreds of electronic equipment to be monitored and controlled, manual construction becomes an arduous and tedious job. These steps when a system is entered incorrectly are no more a big issue when using the FSMtool. Now all the user has to do is enter the new data to the XLS connectivity table and adjust the rules to correspond with the made changes. The FSMtool automation makes this change a quick process compared to the delete incorrect data and enter it again manually method.

There is also the question of personnel, if the person, who has done these manual changes and has been maintaining the system, has a severe accident, retires, is switched to another project. What happens when this person becomes unavailable to do the required job for some reason? If the information needs to be entered manually, there is a long learning curve ahead if the new user needs to build the system. Two weeks of going to courses of the PVSS which are held a couple of times a year. To make the building of this system as simple and automatic as possible is one of the aims of The FSMtool. The project has some dates that can not be delayed, if the system is not tested and ready to go underground on these dates, it may take months or half a year for the next opportunity to arrive. And to be delayed from one of these dates because of one person has an accident or retires does not sound like a possibility.

At the moment the DCS group consists of four persons and two of these persons are in the group as advisors, this does not leave much possibilities if something happens. This might also be the case in other experiments also. When these persons leave, their experience in building and maintaining these systems is also gone, meaning that the next user of the PVSS that comes to design and construct the system has a lot to learn.

By using the FSMtool, the personnel must still be trained, but the time to learn the main tasks is reduced since manual work is reduced.

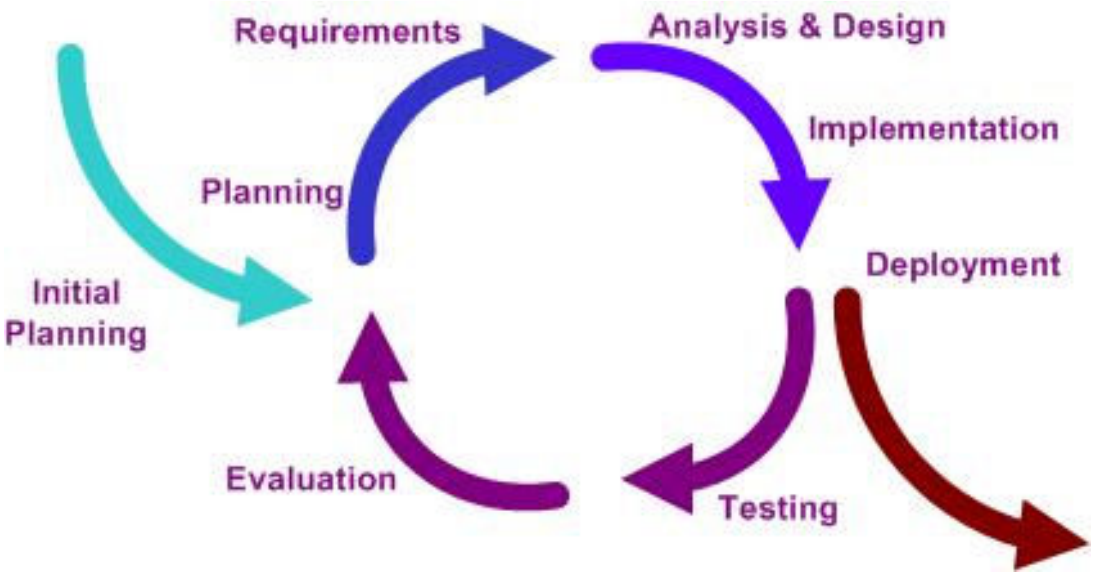


Figure 3.1. Iterative development model used in TOTEM. [3]

The detector hardware system, T1, T2 and the Roman Pots, are developed, tested and deployed in parts. The detector control system has to be ready before the system is tested and fully deployed to the interaction point 5 in CMS. Figure 3.1 shows the development model. This development model instigates one more problem for the user. What if the requirements change and some of the components need to be moved to another logical section or the data points need to be changed due to some error, causing the system to change? What if we restructure the system to be more logically sound? If these changes need to be ready and running the next day, time becomes very essential. When this is the case, The FSMtool makes it possible to enter the changes to the system much more quickly, since these changes are made directly to the XLS tables by the users who wish the changes to be applied to the system.

3.1 Data Platform

The students, professors, engineers and physicists who work for the experiment all come with different backgrounds, some know programming and some do not, most of

them understand and know how to use Microsoft Office products since it is the most used document tool collection. The ones who do not understand programming might need to update information to the system so finding a common platform that can be used by everyone and which is readily available for everyone is required. There are all kinds of data storage types that can hold the necessary information of the system, for example the SQL database. The SQL database system is ideal when data storage is required, but when considering the users who do not have the required understanding and experience to use SQL databases it becomes clear that the database is not a feasible solution at the moment.

The main requirement for the FSMtool is the ease of use and easy maintenance of the data. If the data were be in the SQL format, it would not have the same easy way of updating the information as XLS that is used in the experiment at this time. Using the SQL database as the new data storage media would require that users would need to undergo a course where the information storage process would be taught to them. Even if this new course would take place, another problem arises. How to get the personnel in need of this training to attend it. Still the SQL database is one of the possible future implementations, when the database group of CERN has built and tested the database for the project. The future possibility of an SQL database was kept in mind when designing the FSMtool.

The switch to the SQL database will take time and may not have the same results as with the already implemented and used excel file format. It is quite safe to presume that most, if not all of the staff who work at CERN have some kind of general knowledge on how to use excel tables and if not, it is easy to teach and courses on the matter area also available. The PVSS ctrl scripts understand comma separated values, so it had to be of a type that can easily be transformed into comma separated values. Microsoft Excel has both of these requirements and it is readily available for all users. MAC and Linux systems also understand the excel file format via their own software.

At first the rules were embedded in the programming of the FSMtool but this does not comply with the first requirement of the FSMtool, which is ease of use and maintainability. To maintain these rules the user would need a programming environment, such as Microsoft Visual Studio or Eclipse. After some consideration

the rules of the FSMtool are implemented with XML because the simplicity and understandability of the structure of the XML files, this is because of the structure of XML files are easy to understand. For the XML files there are a lot of material available online so the need for keeping courses on the subject is not necessary when the XML structure is kept as simple as possible.

3.2 Usability and Maintainability Requirements

The program is designed to mainly work as a script, with the PVSS giving it the necessary commands through the command line as the PVSS packages for the T1, T2 and Roman Pots are installed. These packages contain the panels and required information for the control system but they do not have the automation to read from XLS tables. This is where The FSMtool comes in. The command line commands themselves are discussed in the implementation chapter later. If the user sometime needs to use the program itself, a graphical user interface is designed and implemented keeping the ease of use in mind.

The FSMtool is coded using C sharp, an object oriented language developed by Microsoft within the .NET initiative. This also adds up to the maintainability by giving the next developer an easier, structural way of adding new features and updating the existing ones already in the FSMtool. One example is the excel object, which in itself is capable of retrieving the names of the tables in the excel workbook, to copy the sheets to data tables and cleaning up the data tables itself of empty rows and making the excel table resemble more a data table with correct column names. After these steps, the data table itself is the target of the next object and so on. By designing the FSMtool like so, it gives a more clear picture of what the program is doing and gives the programmer a clear view on what it is supposed to do.

4 THEORY

This chapter describes the concepts and theories behind the PVSS supervisory control and data acquisition software and the FSMtool. The PVSS concepts are described briefly as the theory of the finite state machines and the heuristics of the FSMtool are the main subjects.

4.1 The PVSS

The PVSS has a highly distributed architecture. A PVSS application is composed of several processes called managers. These Managers communicate via a PVSS specific protocol over TCP/IP. Managers subscribe to data and this is then only sent on change by the Event Manager, which is the heart of the system. Furthermore, Drivers can be configured to only send data to the Event Manager when a significant change is seen. In a correctly configured system there is essentially no data traffic in stable conditions when the process variables are not changing. [4]

The Event Manager, abbreviated EVM, is responsible for all communications. It receives data from drivers and sends it to the Database Manager to be stored in the data base. It also maintains the process image in memory, the current value of all the data. It also ensures the distribution of data to all Managers which have subscribed to this data. The Database Manager, abbreviated DBM, provides the interface to the run-time data base. User Interface Managers, abbreviated UIM, can get device data from the database, or send data to the database to be sent to the devices, they can also request to keep an open connection to the database and be informed when new data arrives from a device. [4]

There is a user interface for Linux and a Native Vision for Windows. The UIM can also be run in a development mode; PARA for parameterization of data points and data point types, GEDI for the Graphical Editor. Ctrl Managers provide for any data processing as background processes by running a scripting language. This language is like C with extensions, Application Programming Interface managers. This allows users to write their own programs in C++ using a PVSS API to access the data in the

database. Drivers provide the interface to the devices to be controlled. These can be PVSS provided drivers like Profibus and OPC. [4]

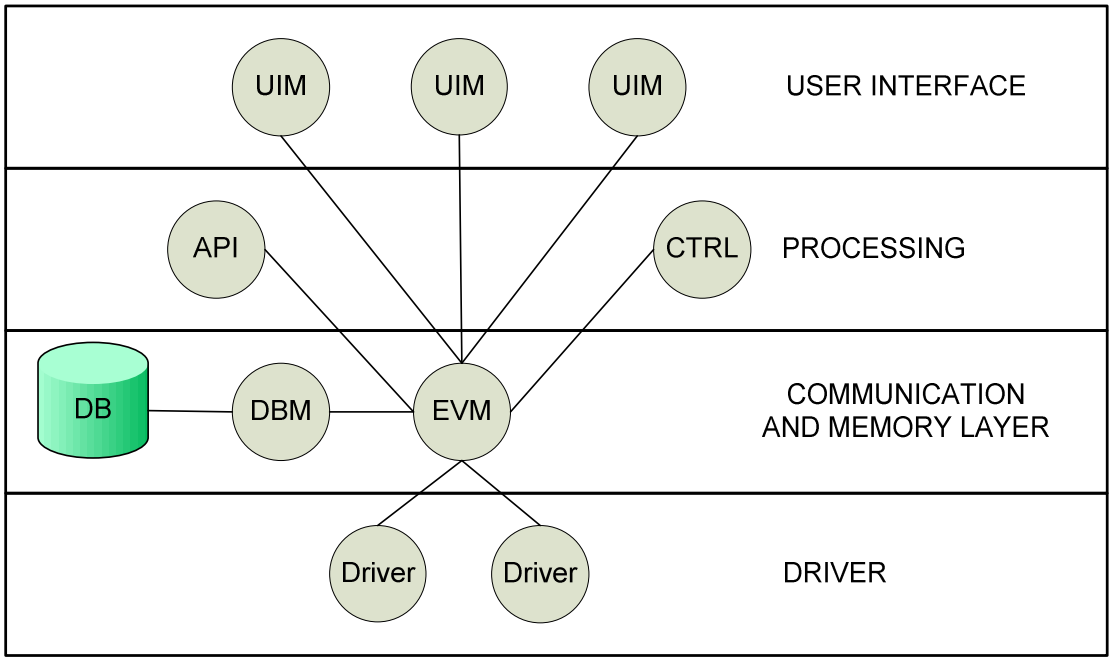


Figure 4.1. A complete layout of the PVSS managers. [4]

Archive Managers allow users to archive data for later access and viewing. A project, which is the name for a PVSS application, may have one or more Archive Managers and one can configure which data is stored in which manager. A complete layout of these managers is shown in figure 4.1. ASCII Manager allows users to export and import the configuration of a project data points and data point types to and from an ASCII file. It allows users to select which data should be exported. The resulting files can be modified, for example using Excel, and re-imported. The import feature can be very useful for transferring data points and data point types from one project to another and it is used when adding Framework components to a project. [4]

All the PVSS applications, as well as the majority of the PVSS tools, are essentially built of panels and scripts. Panels can be created using the Graphical Editor and may include static or dynamic widgets. Dynamic widgets require CTRL scripting to create the required behavior. Typical behavior would be the setting of values to the hardware or the display of data originating from the hardware. [4]

A PVSS System is an application containing one data base manager and one event manager and any number of drivers, user interfaces and so on. The PVSS Managers can run on Windows or Linux and they can all run in the same machine or be distributed across different machines, including mixed Windows and Linux environments. When the managers of one system run distributed across different machines this is called a PVSS Scattered System. [4]

The PVSS can provide for very large applications, in which case one PVSS system would not be enough. In this case a PVSS Distributed System, a group of communicating PVSS systems, can be used. A distributed system is built by adding a distribution manager to each system and connecting them together. multiple systems can be connected in this way. The device data in the PVSS database is structured as so called data points of a pre-defined data point type. The PVSS allows devices to be modeled using these. As such it allows all data associated with a particular device to be grouped together rather than being held in separate variables. [4]

A data point type describes the data structure of the device, data point types are similar to classes in object oriented terminology, and a data point contains the information related to a particular instance of such a device. Data points are similar to objects instantiated from a class in object oriented terminology. The individual parameters are called data point elements and are user-definable. After defining the data point type, the user can then create data points of that type which will hold the data of each particular device. The data point elements mapping to the device data can be of type integer, float, string, and more, for example dynamic arrays of the simple data types, such as dynamic integer, dynamic float and dynamic string. Data point elements can also contain references to other data points or dynamic arrays of such references. [4]

In order to access the data of a particular element of a data point, for example a channel voltage monitor, the user can address it as: Channel1.readings.vMon. The creation and modification of data point types and data points can be done either using the graphical parameterization tool PARA, or programmatically using ctrl scripts. Data points, hardware names, logical tree and the finite state machine create a map between the hardware devices and the detector elements. A map between the hardware

names, also called data point names and the logical names, also called aliases in the PVSS. [4]

The PVSS allows users to design their own user interfaces, in a drag and drop way. For that a special user interface manager, the graphic editor can be used. By using the graphic editor, the user can first design the static part of a panel, by placing elements like buttons, tables and plots. Actions can then be attached to each element. Depending on the element type, actions can be triggered for example on initialization, user click or double click. Actions are programmed using the PVSS scripting language, CTRL scripts. The PVSS provides several drivers to connect to various types of hardware or to different communication protocols. The type of protocol to be used can be configured using the parameterization tool by selecting the address configuration of the data point element and configuring it appropriately. [4]

4.2 XML

In 1986, Standard Generalized Markup Language, abbreviated SGML, was created. SGML defines the descriptions of the structure and content of different types of electronic documents. The applications delivering SGML information over the web convert the data to a hypertext markup language, abbreviated as HTML. Most of the original information of the SGML is lost in the transformation. The lost intelligence removes much of the information flexibility and so creates an obstacle for reusing, interchanging, and automation. This is why Extensible Markup Language, abbreviated as XML, was created. Because of the lack of SGML support in the mainstream web browsers. [5]

XML was created to regain the power and flexibility of SGML without its complexity. XML removes many of the more complex features of SGML that make the authoring and design of software difficult and costly while keeping the beneficial features of SGML. XML is missing some important capabilities of SGML that mostly affect creation of documents but not the delivery of documents. XML was not designed to replace SGML in every aspect. XML is a meta-language, it is a language in which other languages are created. The XML standard was created by W3C to provide an easy to use and a standardized way to store data that describes both its content and its

structure, XML is nothing by itself. XML is more of a "common platform" standard. The main benefit of XML is that it can be used to take data from a program like MySQL, convert it into XML, and then share that XML with other programs and platforms. Each one of these receiving platforms can then convert the XML into a structure the platform uses. [5]

4.3 C# And XML

C sharp and .NET framework have classes to read XML and from these classes the XmlReader class was chosen, since it has the methods and functions needed and it is best suitable for the purpose from all the XML classes. XmlReader class is an abstract base class that provides forward-only and read-only access to the XML data. The class supports reading XML data from a stream or a file. It defines methods and properties that allow moving through the data and read the contents of a node and the current node refers to the node on which the reader is positioned. The reader advances using read methods and properties to return the value of the current node. [6]

The base XmlReader provides only the most essential functionality for reading XML documents. It does not, for example, validate XML. The XmlReader, as its name implies, can only read XML. It is an abstract base class that provides a read-only, forward-only, event based XML pull parser. [5]

An event in a stream based XML reader represents the start or the end of an XML node as it is read from the data stream. The event information is delivered to the application and the application takes action based on that information. In XmlReader events are delivered by querying XmlReader properties after calling its Read method. XmlReader Being forward-only, once a node has been read from an XML document, it is not possible to go back and read it again. Figure 4.2 Shows an example of the use of the XmlReader. [5]

```
public void Process(string source)
{
    //Reset();

    // Set the validation settings.
    XmlReaderSettings settings = new XmlReaderSettings();
    settings.ProhibitDtd = true;

    if (File.Exists(source))
    {
        XmlReader reader = XmlReader.Create(source, settings);

        while (reader.Read())
        {
            if (reader.NodeType == XmlNodeType.Element)
            {
                switch (reader.LocalName)
                {
                    case "include":
                        string filename = reader.GetAttribute("file");
                        filename = Path.Combine(Path.GetDirectoryName(source), filename);
                        filename = Path.GetFullPath(filename);
                        Process(filename);
                        break;
                    case "version":
                        Logger.Record("XML version: " + source + " " + reader.GetAttribute("value"));
                        break;
                    case "PVSSrule":
                        _PvssRuleCollection.Add(ProcessPvssRule(reader.ReadSubtree()));
                        break;
                    case "FSMrule":
                        _FsmRuleCollection.Add(ProcessFsmRule(reader.ReadSubtree()));
                        break;
                }
            }
        }
    }
    else
    {
        Logger.Record("Rules files " + source+ " not found");
    }
}
```

Figure 4.2. XML reader example.

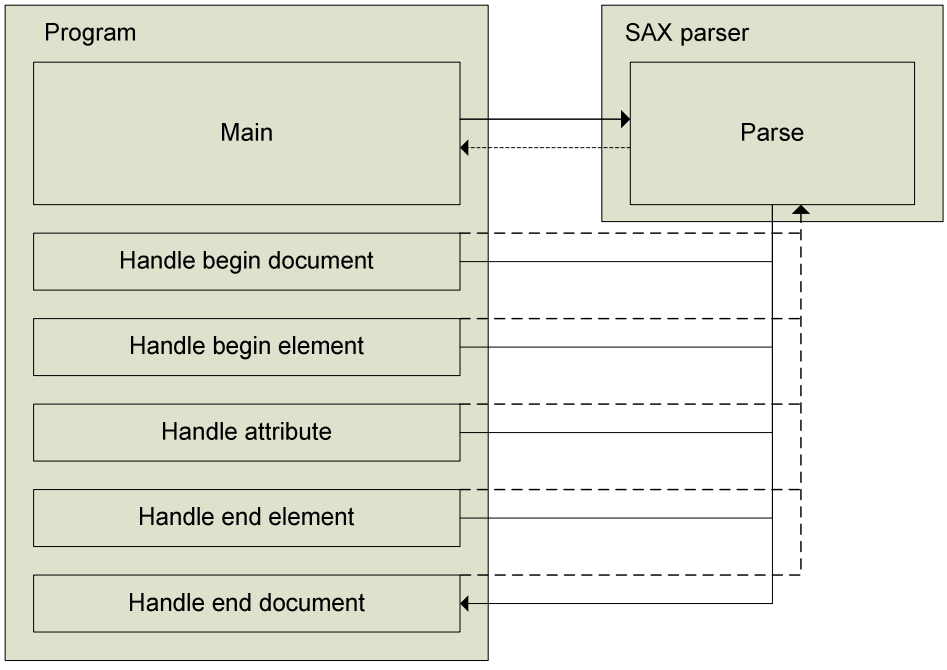


Figure 4.3. XML push parsing model.

In many ways, XmlReader is like the Simple API for XML, SAX. They both work by reporting events to the client. However, there is one difference between XmlReader and a SAX parser. While SAX implements a push parser model, XmlReader is a pull parser. Figure 4.3 shows the push parser model in detail. A push parser requires registering a callback method to handle each event. As the parser reads data, the callback method is dispatched as each appropriate event occurs. Control remains with the parser until the end of the document is reached. [5]

For example, in order to decide on a particular action, it is needed to know how deep the parser is in an XML tree, or be able to locate the parent of the current element. In a pull parser, the code explicitly pulls events from the parser. Running in an event loop, the code requests the next event from the parser. Because of the control of the parser, a program with well-defined methods for handling specific events can be written, and even completely skip over events that are not interesting. A pull parser also enables writing code as a recursive descent parser. This is a top-down approach in which the parser is called by one or more methods, depending on the context. The recursive descent model is also known as mutual recursion. The structure of a program using XmlReader can be very similar to the structure of the XML document it reads. [5]

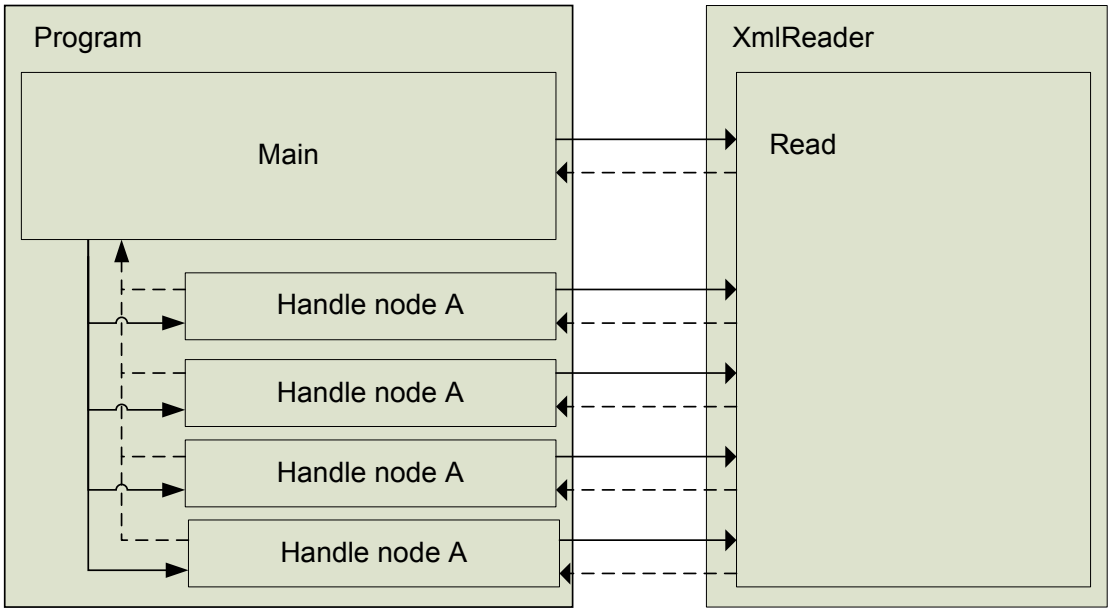


Figure 4.4. XML pull parser model.

Figure 4.4 shows the pull parser in detail. Since XmlReader is a read-only XML parser, it should be used when needed to read an XML file or stream and convert it into a data structure in memory, or when converting it into another file or stream. Because it is a forward-only XML parser, XmlReader may be used only to read data from beginning to end. These qualities combine to make XmlReader very efficient in its use of memory. Only the minimum amount of data required is kept in memory at any given time. [5]

4.4 Regular Expressions

Regular expressions are the key to powerful, flexible, and efficient text processing. Regular expressions themselves, with a general pattern notation almost like a mini programming language, allow describing and parsing text. With additional support provided by the particular tool being used, regular expressions can add, remove, isolate and do a lot more to all kinds of text and data. It might be as simple as a text editor search command or as powerful as a full text processing language. Today a full program that solves the doubled-word problem can be implemented in just a few lines. [7]

A regular expressions so called engine is a piece of software that processes regular expressions, trying to match the pattern to the given string. The engine is part of a larger application and the engine is not accessed directly. The application will invoke it when needed, making sure the right regular expression is applied to the right file or data. [7]

With a single regular expression search and replace command, doubled words can be found and highlighted in the document. With another, all lines without doubled words can be removed. Finally, with a third, each line to be displayed can be ensured that the line begins with the name of the file. The host language, in this case C sharp, provides the peripheral processing support, but the real text handling power comes from regular expressions. Dealing with general text is a problem. Books, program listings, reports, HTML, code, any text imaginable. If a particular need is specific enough, such as selecting files, a specialized scheme or tool can be developed to help accomplish it.

However, a generalized pattern language has been developed over the years, which is powerful and expressive for a wide variety of uses. [7]

Each program implements and uses them differently. In general, this powerful pattern language and the patterns themselves are called regular expressions. Full regular expressions are composed of two types of characters. The special characters, like the "*" from the filename analogy, are called metacharacters, while the rest are called literal, or normal text characters. What sets regular expressions apart from filename patterns are the advanced expressive powers that their metacharacters provide. Filename patterns provide limited metacharacters for limited needs, but a regular expression language provides rich and expressive metacharacters for advanced uses. [7]

It might help to consider regular expressions as their own language, with literal text acting as the words and metacharacters as the grammar. The words are combined with grammar according to a set of rules to create an expression that communicates an idea to find text. Complete regular expressions are built up from small building blocks. Each individual building block is quite simple, but they can be combined in many ways. Knowing how to combine them to achieve a particular goal takes some experience. Probably the easiest metacharacters to understand are "^" and "$", which represent the start and end of the line of text as it is being checked. The regular expression "room" finds "room" anywhere on the line, but "^room" matches only if the "room" is at the beginning of the line. The "^" is used to effectively anchor the match to the start of the line. Similarly, "room$" finds "room" only at the end of the line, such as a line ending with "mushroom". The caret and dollar are special in that they match a position in the line rather than any actual text characters themselves. [7]

There are various ways to actually match real text. Besides providing literal characters like "room" in regular expression, it is possible to also use some of the items described next. The metacharacter "." called dot or point, is a shorthand for a character class that matches any character. It can be convenient when an "any character here" placeholder is needed in the expression. For example, if searching for a date such as "03/19/76", "03-19-76", or even "03.19.76", it is possible to go to the trouble to construct a regular expression that uses character classes to explicitly

allow "/", "-", or "." between each number, such as "03[-./]19[-./]76". However, it is possible to also use "03.19.76". So, "03[-./]19[-./]76" is more precise, but it is more difficult to read and write. "03.19.76" is easy to understand, but not precise enough. [7]

It all depends upon the information about the data being searched, and just how specific the search needs to be. One important, recurring issue has to do with balancing the knowledge of the text being searched against the need to always be exact when writing an expression. For example, knowing that with the data being processed it would be highly unlikely for "03.19.76" to match in an unwanted place, it would certainly be reasonable to use it. Knowing the target text well is an important part of wielding regular expressions effectively. [7]

A very convenient metacharacter is "|", which means or. It allows combining multiple expressions into a single expression that matches any of the individual ones. For example, "Niilo" and "Kauko" are separate expressions, but "Niilo|Kauko" is one expression that matches either. When combined this way, the expressions are called alternatives. For example "grey or gray" can be written as "grey|gray", and even "gr(a|e)y". The latter case uses parentheses to constrain the alternation, parentheses are metacharacters too. Within a class, the "|" character is just a normal character, like "a" and "e". With "gr(a|e)y", the parentheses are required because without them, "gra|ey" means "gra" or "ey", which is not what is needed. [7]

Another example is "(First|1st)•[Ss]treet". Actually, since both "First" and "1st" end with "st", the combination can be shortened to "(Fir|1)st•[Ss]treet". That is not necessarily quite as easy to read, but be sure to understand that "(first|1st)" and "(fir|1)st" effectively mean the same thing. Also, take care when using caret or dollar in an expression that has alternation. [7]

Compare "^From|Subject|Date:•" with "^(From|Subject|Date):•". Both appear similar to our earlier email example, but what each matches, and therefore how useful it is, differs greatly. The first is composed of three alternatives, so it matches "^From" or "Subject" or "Date:•", which is not particularly useful. We want the leading caret and trailing ":•" to apply to each alternative. [7]

4.5 Finite State Machine

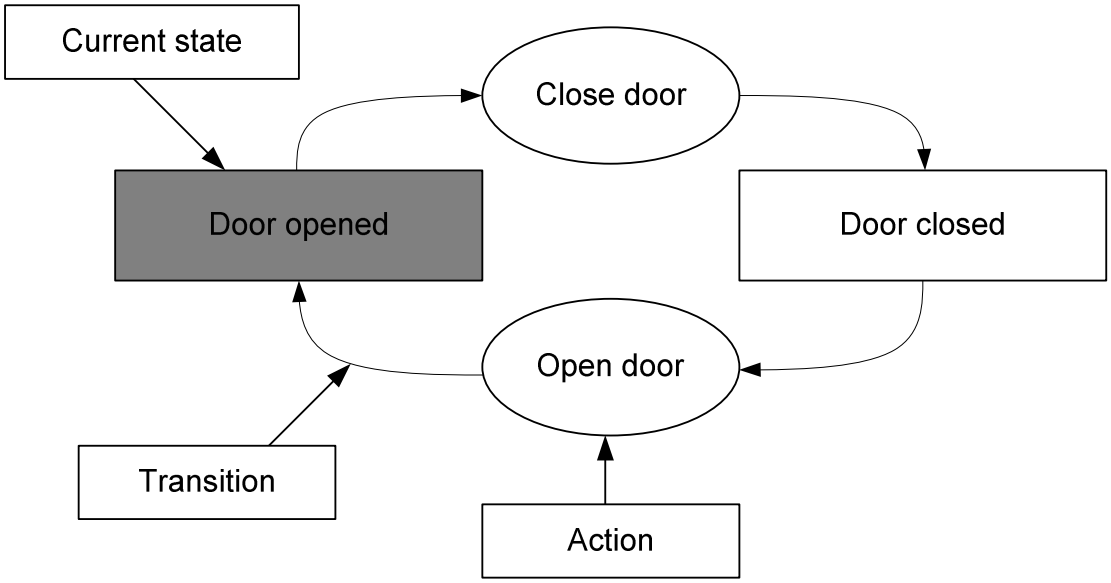


Figure 4.5. FSM door example of different states and transitions.

Figure 4.5 shows an example of the finite state machine. The idea behind the finite state machine is that a system such as a machine with electronic controls can only be in a limited, finite number of states. A window may be open or closed, a coffee machine may be on or off, a CD player may be playing, stopped, paused, rewinding or fast forwarding. These things are referred as systems. Perhaps it is a little strange to call a door a system but not so if designing an automatic door controller.

For an automatic door the operators open and closed are probably insufficient. In that case it might be appropriate to add opening, closing and locked. The first step in any finite state machine design is to identify the significant states of the system. It is required to include all the important states but avoid including unnecessary states. The door may be opening or closing, but half open or 2/3 closed would be unnecessary. Finite state machine control unit is a software entity that monitors the states of its children and reports an overall state to its parent.

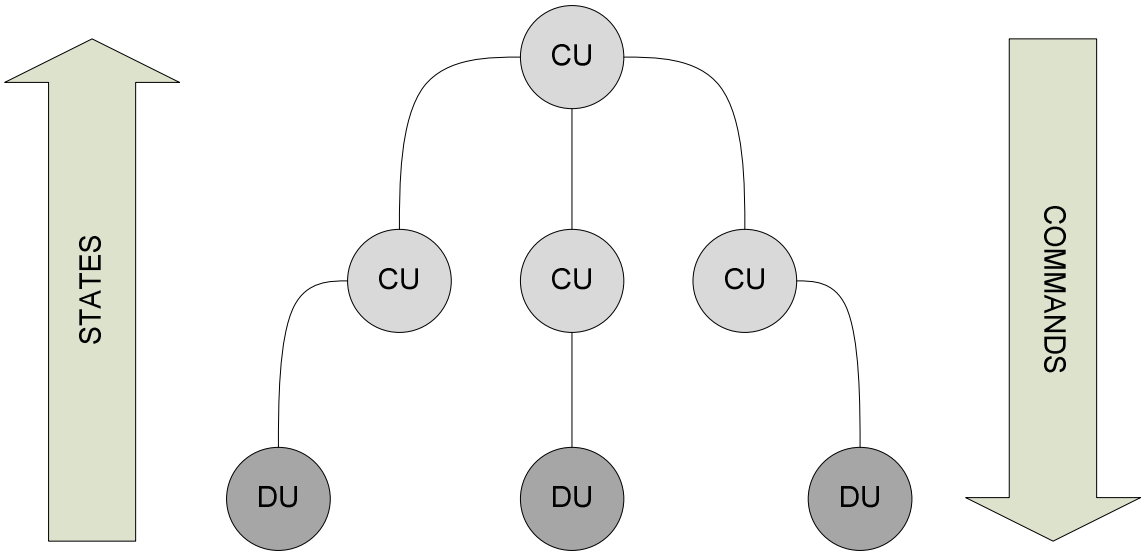


Figure 4.6. Finite state machine hierarchy.

FSM device unit is a software entity that monitors the PVSS data points that represent some hardware. It too reports some overall state to its parent. The device unit is a map between the hardware state and the FSM state. An example of the hierarchy can be seen in figure 4.6. Device units and control units are typically arranged in a tree structure. Device units will always appear as leaves of the tree. The structure nodes of the tree are control units. The control units and device units not only monitor states and report upwards. They also accept commands from user interface panels and their parent control unit. They may also send commands down to their own children. At the lowest level, these commands arrive at the device units and are passed to the hardware. [4]

All of the device units are at the bottom layer of the tree. A control unit at one level in the hierarchy may very well adopt a different set of states to a control unit at another level in the hierarchy. At one level, notions of ON and OFF may be used but at another level, notions of ready or not ready may be used instead. A child node must always communicate with its parent. Sibling or children control units may be of different control unit types and each must use an agreed interface for the communication with the parent. The state of the device unit propagates up to the control unit and if the device unit state is in error then the state of the control unit becomes in error also if the state is allowed to propagate. This propagation of states

allows the user to get notified in the main control panel of the system when there is something wrong for example in the cooling of a GEM detector plane. [4]

4.6 Agile Software Development

Agile software development has its own vocabulary. This vocabulary imposes several important concepts into informal descriptions. There are multiple ways of expressing the same concept in source code. Refactoring is the process of changing the structure of code, rephrasing it, without changing its meaning or behavior. It is used to improve code quality, to fight off software entropy and to ease adding new features. If the software has a bug and it needs to be fixed, there are two kinds of solutions. The right way and the fast way. Choosing the fast way means also ignoring that the code is made a bit messier. Technical debt is the total amount of less-than-perfect design and implementation decisions in the project. [8]

This includes quick hacks intended just to get something working right now and design decisions that may no longer apply. Technical debt can even come from development practices such as an incomplete test coverage. It is in enormous methods filled with commented-out code and TODO how this works comments. [8]

The bill for this debt often comes in the form of higher maintenance costs. Simple tasks that ought to take minutes may stretch into hours or afternoons. Left unchecked, technical debt grows to overwhelm software projects. It is not wise to rewrite the software, but it happens all the time. Unchecked technical debt makes the software more expensive to modify than to re-implement. The key to managing it is to be constantly vigilant. Avoid shortcuts, use simple design, refactor, refactor and refactor some more. [8]

Some activities invariably stretch to fill the available time. There is always a bit more polish that can be put on a program or a bit more design to discuss in a meeting. At some point a decision has to be made. At some point all options have been identified. If using timeboxing, set aside a specific block of time for research or discussion and stop when the time is up, regardless of progress. This is both difficult and valuable. It is difficult to stop working on a problem when the solution may be seconds away.

However, recognizing when as much progress as possible is made is an important time-management skill. Timeboxing meetings, for example, can reduce wasted discussion. If commitments are delayed beyond the last responsible moment, then decisions are made by default, which is generally not a good approach to making decisions. By delaying decisions until this crucial point, an increase in the accuracy of decisions can be seen, decrease workload, and decrease the impact of changes. [8]

Stories represent self-contained, individual elements of the project. They tend to correspond to individual features and typically represent one or two days of work. They are not implementation details, nor are they full requirements specifications. They are traditionally just a paper note of information used for scheduling purposes. [9]

An iteration is the full cycle of designing, coding, verifying and releasing. It is a timebox that is usually one to three weeks long. Each iteration begins with the customer selecting which stories the team will implement during the iteration and it ends with the team producing software that the customer can install and use. The beginning of each iteration represents a point at which the direction of the project can be changed. Smaller iterations allow more frequent adjustment. Fixed-size iterations provide a well timed rhythm of development. Though it may seem that small and frequent iterations contain a lot of planning overhead, the amount of planning tends to be proportional to the length of the iteration. [9]

In well-designed systems, programmer estimates of effort tend to be consistent but not accurate. Programmers also experience interruptions that prevent effort estimates from corresponding to calendar time. Velocity is a simple way of mapping estimates to the calendar. It is the total of the estimates for the stories finished in an iteration. In general, the team should be able to achieve the same velocity in every iteration. This allows the team to make iteration commitments and predict release dates. velocity is a technique for converting effort estimates to calendar time and has no relation to productivity. [9]

Regardless of how much work testers and customers do, many software teams can only complete their projects as quickly as the programmers can program them. If the rest of the team outpaces the programmers, the work piles up, falls out of date and

needs reworking and slows the programmers further. Therefore, the programmers set the pace, and their estimates are used for planning. As long as the programmers are the constraint, the customers and testers will have more slack in their schedules, and they will have enough time to get their work done before the programmers need it. Legacy projects in particular sometimes have a constraint of testing, not programming. The responsibility for estimates and velocity always goes to the constraint: in this case, the testers. Programmers have less to do than testers and manage their workload so that they are finished by the time testers are ready to test a story. If testers are the constraint, programmers might introduce and improve automated tests. [9]

Agility, the ability to respond effectively to change, requires that everyone pay attention to the process and practices of development. This is mindfulness. Sometimes pending changes can be subtle. The technical debt is starting to grow when adding a new feature becomes more difficult this week than last week. [8]

4.7 Object Oriented Programming

Many languages, such as C++ and Java, are said to support objects, but few languages actually support all the principles that constitute object-oriented programming. C sharp is one of these languages and it is designed from the ground up to be a object-oriented, component-based language. C++ is not, because of the fact that its roots are deeply in the C language. Too many object-oriented ideals had to be sacrificed in C++ to support legacy C code. Even the Java language, as good as it is, has some limitations as an object-oriented language. Object-oriented programming is an entire set of concepts and ideas. It is a way of thinking about the problem being addressed by a computer program and dealing with the problem in a more intuitive and productive manner. [10]

Object-oriented programming is a different way of thinking about how to design and program solutions to problems. The benefits of programming with an object-oriented language are writing code more efficiently and having a system that can be easily modified and extended once written. In a object-oriented language the problems are expressed through objects. Encapsulation is one of the main principles of object-

oriented programming, the ability to hide the internal data and methods of the object. The ability to present an interface that makes the object accessible programmatically. How an object carries out its job is not important as long as the object can perform the given job. An interface is given to the object and that interface is used to make the object perform the tasks that are required from it. [10]

The differences between a class and an object is confusing for programmers who are new to the terminology of object-oriented programming. People who had no prior coding experience were found to learn object-oriented programming more easily. An object can be distinguished from a class in many ways. A class can be thought as a type that has methods. An object on the other hand is an instance of a type or a class. A unique term to object-oriented programming called instantiation is basically the act of creating an instance of a class. The created instance is then called an object. After the instantiation, the communication with this object is done through its public members. [10]

Most of the object-oriented languages support the definition of an array of objects. The ability to create an array of objects grants the programmer the possibility to easily group objects and iterate through objects by methods of the object array. Compare this to a linked list where each item in the list is linked manually to the items that comes before and after the linked item. Encapsulation is the boundary between a class external interface and the internal details. The internal, public members of the class are made available to the external while keeping the internal members secure. [10]

The advantage of encapsulation of the class is that it can be exposed while hiding and keeping secure the class internals. It is possible to create or derive a new class that is based on an already existing class through inheritance. This makes it possible to modify the class the way that is needed and to derive new objects. This is the core of creating a class hierarchy. Besides abstraction, inheritance is the most significant part of designing a new system. Deriving a class means that a new class is derived from the base class that inherits all the members of the base class. Derived class enables the reuse of the old class and the work that came with it. [10]

The last thing to do in object oriented programming is to burden the users of the class with the need to understand all the internal details of the class design. To burden the

users with this knowledge would be like the manufacturer of engines requiring the customers to understand the internal mechanics of the engine before a car could be purchased. Using the code like this does not promote reuse. To avoid this simply define all of the members for the base class so that they would function the same regardless of the type of the engine. The derived class would then inherit this functionality and change it if necessary. Inheritance enables the reuse of the code by inheriting the functionality from the base classes allowing the user to extend the functionality of the class by adding new variables and methods. [10]

Polymorphism can be defined that it is functionality that allows old code to call new code. This can be counted as the biggest benefit of object-oriented programming. Polymorphism allows extending or enhancing the code without modifying or breaking the existing old code. [10]

With polymorphism comes at least two beneficial abilities. The first advantage is that it gives the ability to group objects that share a common base class and treat the objects consistently. Polymorphism will ensure during runtime that the correct method of the derived object is called. The second advantage being the reuse of old code. [10]

5 IMPLEMENTATION

This chapter describes the designing and implementation of the FSMtool. The decisions and the reasons why the FSMtool is built like this. The menu structure of the FSMtool and what each of the functions of the FSMtool. The use cases describe the functionality of the programs graphical user interface. The XML part of this chapter describes how the FSMtool works with the files. At the end are the commands that can be given through the command prompt.

5.1 Implementation and Design

As stated in chapter 3, previously the user of the PVSS program has entered all the data manually. This amounts to a lot of tedious work that could be done automatically. The question, which led to the point that the tool is implemented: "See if it is possible to convert XLS tables programmatically into CSV files with some programming language". A brief study of different programming languages took place and it was decided after some consideration that C# holds the best suited functions and interoperability with XLS files. C# is an object oriented language and this helps in keeping the program at hand in shape. The FSMtool is programmed with the theories of object oriented programming in mind. This helped in keeping the FSMtool usable after each revision.

It was decided to go with agile development methods and include the program into the main project plan of the TOTEM DCS group. The basis is that the program had to be usable after each stage so agile methods were the obvious choice. Since with agile methods the direction of the program could be changed during each weekly meeting with the group.

The next step was to study if and how it is possible to manipulate the XLS files and store them in the program to be able to create new table content with rules. C# functions that could manipulate the XLS files were considered but it is not a good option to take since it would limit the accessibility to the data. It was decided that data tables are the right option since they can be easily manipulated and stored in the program.

Now the program could store the data into data tables but the data needed some modification functions because the data is not in a usable form. All the data is stored in a data table but it had no way of identification for columns. After some correcting functions were implemented for the data it is showing some promise. Now the data could be handled with the column names.

The next phase was to study how to implement the rules that could change the data in the tables. First were the tryouts to see how the rules could get a hold of the correct data in the tables, this took some time and the rules were first implemented in the programming itself. Regular expressions were suggested by my tutor and after some testing they were made to work and the correct data could be found with the rules.

The rules were now in the code but this posed a problem considering the future. If the rules needed to be changed by someone who does not know C# they could not do it without help. So a brief study of which format the rules could be stored took place. After some consideration it was decided that XML files could be the answer. After studying XML, its structure and how to read XML with C# the first working rule is made in a single file.

After implementing tens of rules to the XML file it became clear that one file is growing too large and difficult to maintain and find the correct rules. So the rules were divided each into its own file by what table and sheet did the rules apply to. The main XML file contains the information of all the rule files by sheet name and if it is either pinout or finite state machine file.

The FSMtool could now read the data into internal structure, make it more database like and apply rules located in the XML files to the data. One phase remained still, how to create the CSV file that holds the finite state machine so that the PVSS scripts understand how to read them. This particular CSV file is formatted differently than the pinout files. This formatting posed a problem since the indent increase in the CSV file showed the place in the hierarchy in the finite state machine. This problem was solved using a tree structure in the program that is built using the finite state machine rules and the tree is then exported into CSV file and the tree structure itself holds the level of the indent.

The PVSS understands command line prompt and can write into the command line prompt. A command interface for the FSMtool was implemented to allow PVSS scripts to work with the FSMtool. This allowed the PVSS to start the FSMtool as a process and give it the necessary commands through the command prompt. A graphical user interface is implemented for the program to help in developing the program and the rules for the FSMtool.

The final version of the FSMtool holds the rules in XML so that the average user of the program does not have to understand programming, only to understand XML and regular expressions. This way the program can be used with the graphical user interface to test the rules within the FSMtool and ease the modification of the rules. Some may consider why the data is implemented to data tables and not into, for example, SQL table format directly. This is taken into consideration as well, and even suggested, but for the data table to be used it has to be approved by the CERN data management group to be included in the main data center. Because the SQL tables were not an available option at the time, an option to include the tables is left to the program.

5.2 Use Cases

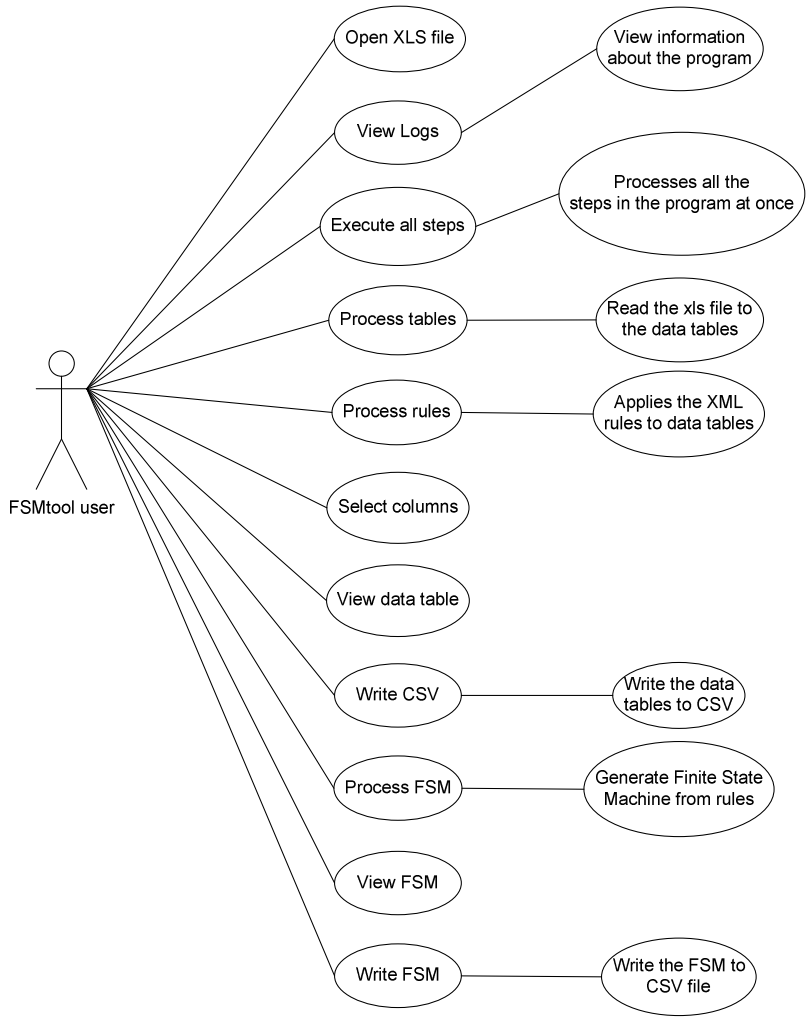


Figure 5.1. The FSMtool main menu

Through the main menu the user is able to perform the same tasks manually as through the command prompt. The Graphical user interface is implemented for testing the latest rules and so that it can be used as a separate program without the help of the PVSS scripts. The Graphical user interface makes it more simple to handle the rules and to perform simple tasks in the XLS files such as selecting one single column of the XLS file to be viewed or exported to a CSV file. Figure 5.1 shows the use case diagram of the main menu.

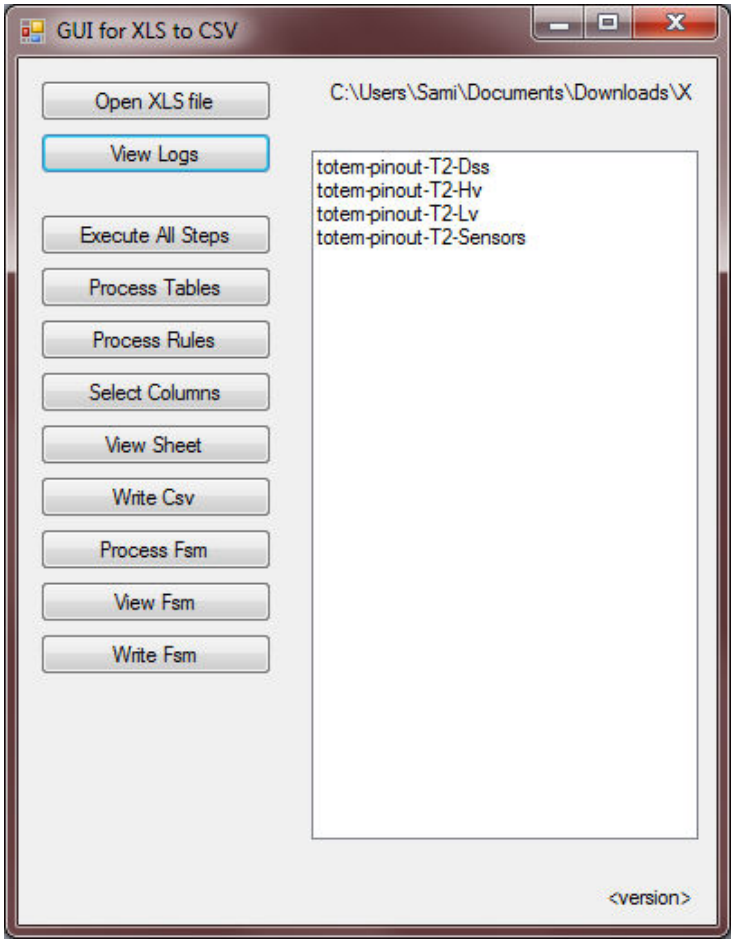


Figure 5.2. Main menu when a XLS table is opened and processed.

Figure 5.2 shows the look of the main menu when a table has been opened with the open XLS file button and the tables have been processed with the process tables button. The sheets are now in internal data table structure and the sheet names are shown on the right hand window. On the down right corner is the version numbering, in this case it just shows <version>, this is because the Subversion version control system is not available at the PC this picture is taken.

5.2.1 Open XLS File

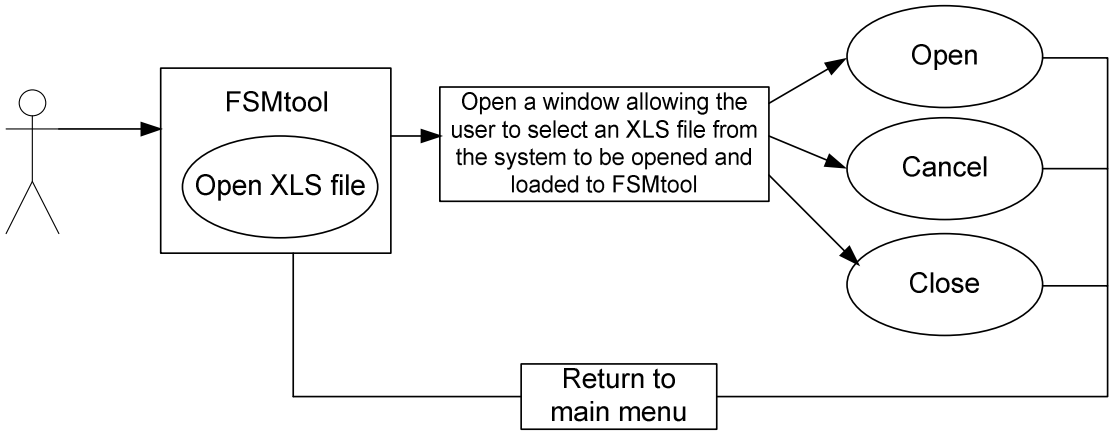


Figure 5.3. Open XLS file case.

The open XLS file is implemented with a .NET framework component that opens a normal windows 'open a file' window. Figure 5.3 shows the use case diagram of opening an XLS file. The user browses to the file location and the options are to open, cancel or close the window. Only by opening a file the window returns the path of the file to the program.

5.2.2 View Logs

The view logs option in the menu opens a window that includes all of the information the FSMtool is doing. The log screen shows the starting execution time, configuration file location, input file location, output folder location and the processes that are being done. Figure 5.4 shows the format of the log window. If the XLS table already contains information, for example if the table already holds a DCS logical name but it is different than the rules state in the program, a notification in the log is written stating the difference. This helps when creating the rules, if the rule is just overwritten without any prompt, something could go wrong. This enables checking the rules and see which one holds the correct statement. This window basically shows the information that is written also to a log file every time the FSMtool has run.

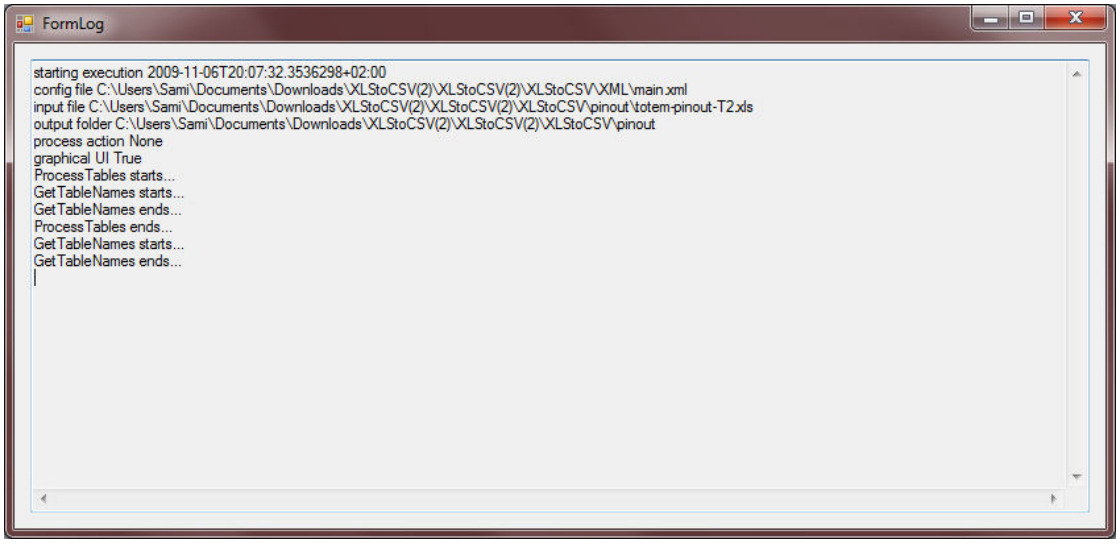


Figure 5.4. Log window.

5.2.3 Execute All Steps

The execute all steps option in the menu does every step that the program is designed to do. These steps include editing the XLS table to a more database-like structure to an internal data table structure, applying the rules described in XML to the data, writing the new data to CSV files, creating the finite state machine model and writing it to a CSV file.

5.2.4 Process Tables

The process tables option reads the XLS sheets and transforms them to a more database-like structure to an internal data table structure. This step is necessary because the XLS sheets contain information that is not required in the data tables like empty columns, empty lines and weird long sheet names that the excel gave for some unknown reason, this way the size of the data tables could be reduced and the way to apply the rules to the data is made easier. The way this is done is that in the XLS tables there is an H marking the header line that holds all of the column names in the sheet and these were made into the column named of the internal data table structure. Then the data is read line by line and added to the tables, after this step all the empty columns and lines were deleted in the data table.

5.2.5 Process Rules

Process rules option applies all the rules in the XML files to the internal data tables of the FSMtool and the identification inside the XML files were done with regular expressions, more on these subjects can be read in the theory chapter.

5.2.6 Select Columns

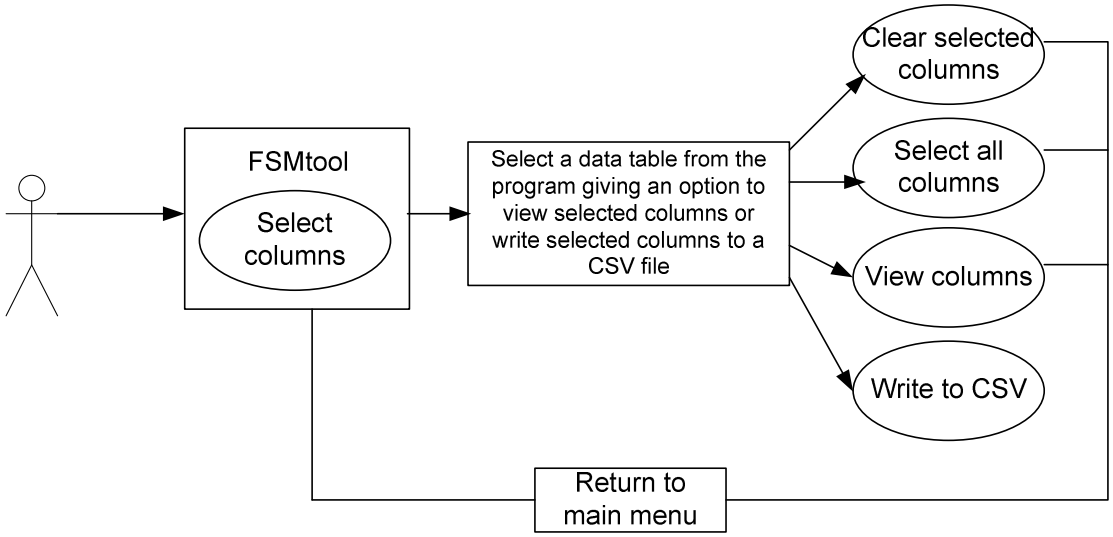


Figure 5.5. Select columns case.

The select columns option is implemented to give the user more simplified way to view the tables. Figure 5.5 shows how the menu functions work and figure 5.6 shows how the window looks. The purpose is to be able to select columns from the table and export them to CSV files, this way the columns needed can be printed to a smaller table of paper when going underground to do wiring and checking connectivity.

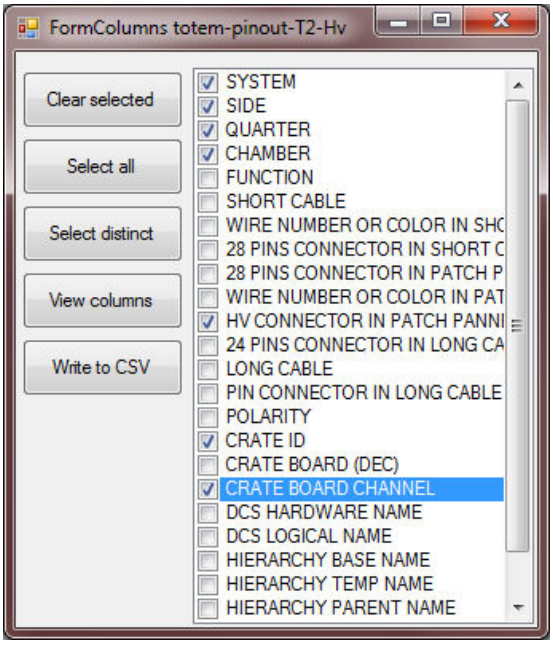
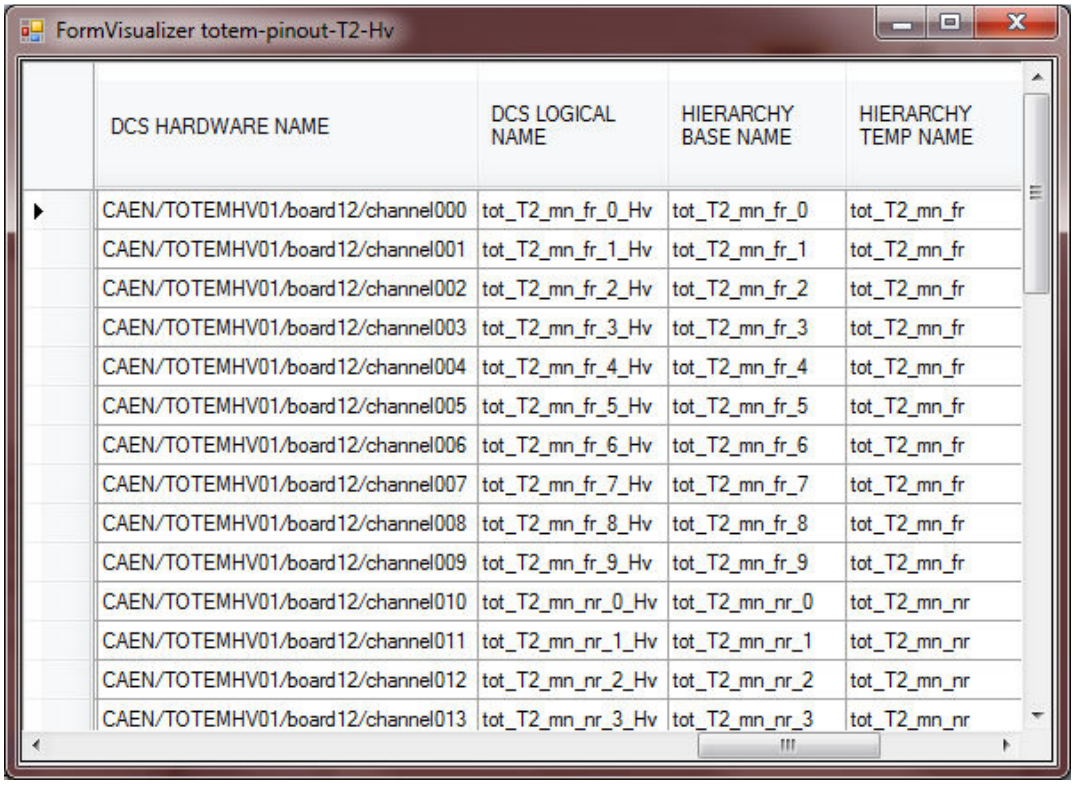


Figure 5.6. Select columns window.

The user is given the option to clear all selected columns, as a default all of the columns are selected. To select all columns, view the selected columns in a form table to confirm that the selected columns hold the data that is wanted and finally to write the data to a CSV file. This CSV file can then be printed instead of the whole sheet of XLS table.

5.2.7 View Sheet

The view sheet option is made so that the user can check the data on the sheet and see if the rules have actually done what they were meant to do. Figure 5.7 it shows that the DCS and hierarchy names were properly generated with the rules. This is a much easier way to view the rules than to write them to a file and then open it to view the changes. The XML files can be edited on the fly with this option, the program does not need to close.



DCS HARDWARE NAME	DCS LOGICAL NAME	HIERARCHY BASE NAME	HIERARCHY TEMP NAME
CAEN/TOTEMHV01/board12/channel000	tot_T2_mn_fr_0_Hv	tot_T2_mn_fr_0	tot_T2_mn_fr
CAEN/TOTEMHV01/board12/channel001	tot_T2_mn_fr_1_Hv	tot_T2_mn_fr_1	tot_T2_mn_fr
CAEN/TOTEMHV01/board12/channel002	tot_T2_mn_fr_2_Hv	tot_T2_mn_fr_2	tot_T2_mn_fr
CAEN/TOTEMHV01/board12/channel003	tot_T2_mn_fr_3_Hv	tot_T2_mn_fr_3	tot_T2_mn_fr
CAEN/TOTEMHV01/board12/channel004	tot_T2_mn_fr_4_Hv	tot_T2_mn_fr_4	tot_T2_mn_fr
CAEN/TOTEMHV01/board12/channel005	tot_T2_mn_fr_5_Hv	tot_T2_mn_fr_5	tot_T2_mn_fr
CAEN/TOTEMHV01/board12/channel006	tot_T2_mn_fr_6_Hv	tot_T2_mn_fr_6	tot_T2_mn_fr
CAEN/TOTEMHV01/board12/channel007	tot_T2_mn_fr_7_Hv	tot_T2_mn_fr_7	tot_T2_mn_fr
CAEN/TOTEMHV01/board12/channel008	tot_T2_mn_fr_8_Hv	tot_T2_mn_fr_8	tot_T2_mn_fr
CAEN/TOTEMHV01/board12/channel009	tot_T2_mn_fr_9_Hv	tot_T2_mn_fr_9	tot_T2_mn_fr
CAEN/TOTEMHV01/board12/channel010	tot_T2_mn_nr_0_Hv	tot_T2_mn_nr_0	tot_T2_mn_nr
CAEN/TOTEMHV01/board12/channel011	tot_T2_mn_nr_1_Hv	tot_T2_mn_nr_1	tot_T2_mn_nr
CAEN/TOTEMHV01/board12/channel012	tot_T2_mn_nr_2_Hv	tot_T2_mn_nr_2	tot_T2_mn_nr
CAEN/TOTEMHV01/board12/channel013	tot_T2_mn_nr_3_Hv	tot_T2_mn_nr_3	tot_T2_mn_nr

Figure 5.7. Logical and hierarchy names properly generated.

5.2.8 Write CSV

The write CSV option in the menu does what it says, it converts the FSMtool internal data table structure to CSV files. CSV is the file type, which the PVSS command scripts understand to read.

5.2.9 Process FSM

Process FSM option generates the FSM inside the program with the rules written in the XML files. The FSM is generated as a tree structure and it is the logical operation model of the control system.

5.2.10 View FSM

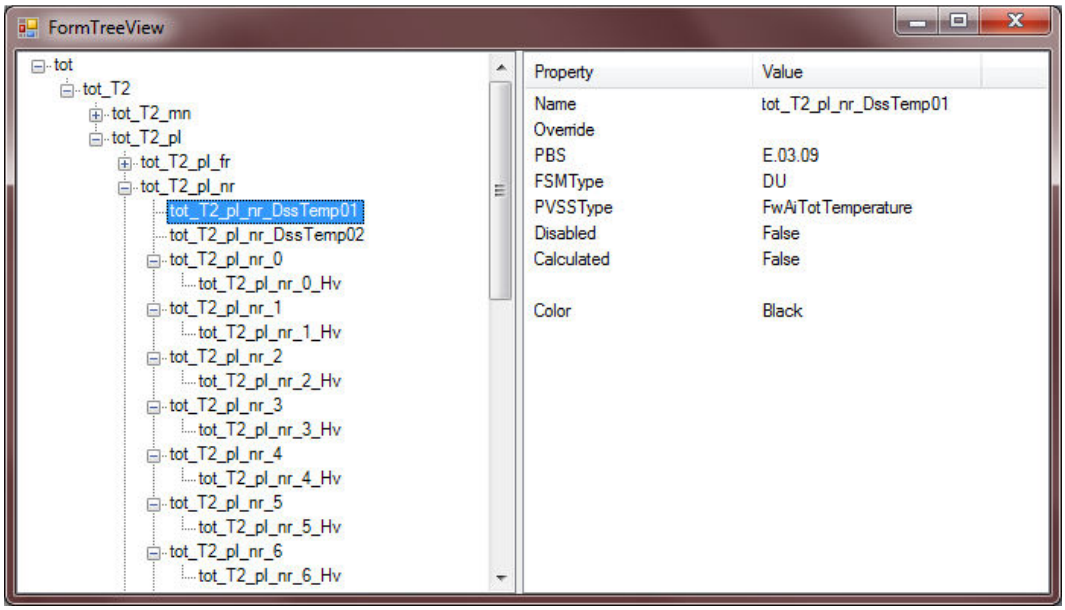


Figure 5.8 View FSM window.

The view FSM option in the menu opens a window shown in figure 5.8. This view enables the user to see if the finite state machine rules in the XML files are working correctly and see if the machine is generated correctly. The windows is split on two parts. The left hand window shows the structure of the FSM tree and the right hand window shows what data the selected node contains.

5.2.11 Write FSM

The write FSM option writes the created FSM into a CSV file, this file has to be exactly correct so that the levels of the FSM model are the same as in CSV. For example tot_T2_pl_nr_DssTemp01 has to be five levels in the file. Tot is the first level in the hierarchy, the top node. DssTemp01 is five levels down in the hierarchy. The levels are empty fields in the CSV file.

5.3 XML Implementation

```
<?xml version="1.0" encoding="utf-8" ?>

<rules>
  <version value="0.1"/>

  <FSMrule>
    <regex expression="^tot_T2$"/>
    <data override="" pbs="E.03.99" typepvss="totT2Sv" typefsm="CU" name="" label="" panel=""/>
  </FSMrule>
  <FSMrule>
    <regex expression="^tot_T2_(pl|mn)$"/>
    <data override="" pbs="E.03.99" typepvss="totT2Side" typefsm="CU" name="" label="" panel=""/>
  </FSMrule>
  <FSMrule>
    <regex expression="^tot_T2_(pl|mn)_(fr|nr)$"/>
    <data override="" pbs="E.03.99" typepvss="totT2Quarter" typefsm="CU" name="" label="" panel=""/>
  </FSMrule>
  <FSMrule>
    <regex expression="^tot_T2_(pl|mn)_(fr|nr)_\d$"/>
    <data override="" pbs="E.03.01" typepvss="totT2Chamber" typefsm="CU" name="" label="" panel=""/>
  </FSMrule>
  <FSMrule>
  <regex expression="^tot_T2_.+_Hv$"/>
  <data override="" pbs="E.03.01" typepvss="FwCaenChannelT2" typefsm="DU" name="" label="" panel=""/>
  </FSMrule>
  <FSMrule>
```

Figure 5.9. XML file example.

Because there are different excel tables and within them multiple sheets of connectivity information, it was decided that the rules should be in different files each for one excel table only. This way the XML rules are more manageable and easier to read. This helps in locating the rules and updating them. If the user wants to modify the finite state machine rules for the roman pots, all that is needed is opening a file named fsm-rp.xml. Figure 5.9 shows the structure of the a XML file that holds the rules and figure 5.10 shows the method for reading the XML files.

An example of a rule is one for the roman pot planes, it is built so that it has all the information required for the roman pot planes. The rule contains the information necessary for the finite state machine, for example the FSM type is a control unit, abbreviated as CU. The FSMtool runs through the data tables and with the help of the rules adds the necessary information to the tables. From the previous chapter this rule can be understood. "^" metacharacter means that the rule begins from the start with **tot_Rp_** and then with the "+" character looks for any number of previous items meaning that any length of characters can be found here but the last characters must be **bt**, **hr** or **tp** at the end of the line. These rules make the maintaining of the tables and files a much easier task.

```
private FsmRule ProcessFsmRule(XmlReader reader)
{
    FsmRule result = new FsmRule();

    while (reader.Read())
    {
        if (reader.NodeType == XmlNodeType.Element)
        {
            switch (reader.LocalName)
            {
                case "regex":
                    result.Expression = reader.GetAttribute("expression");
                    break;
                case "data":
                    result.Override = reader.GetAttribute("override");
                    result.PBS = reader.GetAttribute("pbs");
                    result.TypePvss = reader.GetAttribute("typepvss");
                    result.TypeFsm = reader.GetAttribute("typefsm");
                    result.Name = reader.GetAttribute("name");
                    result.Label = reader.GetAttribute("label");
                    result.Panel = reader.GetAttribute("panel");
                    break;
            }
        }
    }

    return result;
}
```

Figure 5.10. ProcessFsmRule method.

5.4 Commands

The commands can be given through the command prompt to the program without launching the graphical user interface. The command prompt is used because the PVSS understands command prompt and can give commands to the program through it. As previously mentioned, the main purpose of the graphical user interface is giving the same commands that can be given through the command prompt. The purpose of the user interface is to extend the viewing possibilities of the program. Main commands, the core functionality of the FSMtool is in command prompt. The FSMtool is designed to work mainly through its command prompt.

```
FSMtool.exe../../pinout/totem-pinout-T2.xls --gui -c
../../XML/main.xml
```

`{Program name}{Location of the xls file}{commands}`("`../`" means one folder up from the executable).

This is one example of the order in which the commands can be given. The order of the commands matters only when commands --output, --config or --action is given since the command is in two parts.

The following commands are the action commands that can be given in pair with "--action" or "-a" parameter. After the "--action" parameter the following parameters can be given: Basic, pinout or fsm. Basic command converts the given xls table to CSV file format, each sheet of the xls in different CSV files. Pinout parameter Applies the rules to the data and converts the xls file to CSV format. FSM parameter applies the finite state machine rules to the data, creates FSM structure and exports the data to CSV format. The following example is of the finite state machine.

```
FSMtool.exe ../../pinout/totem-pinout-T2.xls -c ../../XML/main.xml -a
fsm
```

The "-v" or "--version" command shows the version of the FSMtool. The "-h" or "--help" parameter show help for the user and how to use the FSMtool. The "-g" or "--gui" command opens the graphical user interface. The "-c" or "--config" parameter defines the path where the XML rule file is located. The "-o" or "--output" parameter

defines the output folder for the CSV files generated. By default this is the same folder as the xls file has, but it can be changed with this command.

```
FSMtool.exe ../../pinout/totem-pinout-T2.xls -a basic -o C:/Temp/CSV/
```

6 CONCLUSIONS

Some afterthoughts on the project and on the TOTEM project. The project gave me a lot of new information both on programming and on distributed control systems. Learning all of the required information and to understand the underlying problem was challenging and at the same time interesting. The PVSS has its own quirks and nooks to learn, most of which became clear in the PVSS courses. It was a pleasure to work with my supervisor and the other team members.

The FSMtool was a success in the field. It has reduced a lot of work that had to be done manually before. What was previously done in the PVSS system by hand is now done in an automated fashion via the FSMtool. This helped the DCS team to reduce the workload when changes to the infrastructure of the system had to be done. Also the number of errors in the creation of the system were reduced since the data is now handled automatically. From a user point of view the rules that create the names and logical connectivity for the system are managed easily through the XML. The project also gained interest within CERN.

The future implementations of the FSMtool could hold a network connected data storage, either in the excel format or SQL format. The implementation of a SQL database could be one of the future improvements. It was considered and found not yet feasible. The database group at CERN had not yet approved and fully tested the database for the TOTEM. This would have been an improvement in the program and its manageability since the excel format is not originally intended for database use. The rules are now implemented in XML, but they are not supervised in any way. A way to check that the rules are correct before applying them could also be one of the future implementations. This might become problematic since the PVSS does not give any feedback to the program besides the commands and all of the logical and hardware names are the result of the team deciding what they are. The FSMtool could also be a part of the PVSS packages at CERN, distributing it to all users giving them a choice to automate a time consuming part of the control system building.

REFERENCES

1. CMS map. September 2009. [online, web-page]. http://www.interactions.org/cms/?pid=2100&image_no=CE02491.
2. TOTEM public webpage. September 2009. [online, web-page]. http://totem.web.cern.ch/Totem/
3. Hung V. Software development process. January 2010. http://cnx.org/content/m28927/latest/ [online, web-page].
4. PVSS introduction to newcomers. September 2009. http://lhcb-online.web.cern.ch/lhcb-online/ecs/PVSSIntro.htm [online, web-page].
5. Elliotte R. H., W. Scott Means. *XML in a Nutshell, 3rd edition*. O'Reilly. 2004.
6. Hoang L., Thuan L. T. .*NET Framework Essentials, 3rd Edition*. O'Reilly. 2003.
7. Regular-expressions.info. Regular Expression Tutorial. September 2009. [online, web-page]. http://www.regular-expressions.info/tutorial.html.
8. Techbookreport.com Agile development in 30 seconds. November 2009. [online, web-page]. http://www.techbookreport.com/tutorials/agile-30-secs.html.
9. Craig L. *Agile and Iterative Develoment: A Manager' s Guide*. Addison Wesley. 2003.
10. Jesse L., Donald X. *Programming C# 3.0, Fifth Edition*. O'Reilly. 2007.