Advances in Computer Science and its Applications
Vol. 1, No. 1, March 2012

www.worldsciencepublisher.org

XML driven SCPI interpreter

[1]Vikas Deshmane

[1] Pune University, Pune, India

Email: deshmane.vikas@gmail.com

Abstract – In 1990, Standard Commands for Programmable Instruments (SCPI) was defined with the IEEE 488.2 specifications. SCPI created a consistent programming model across all manufacturers and models. Today, SCPI is supported by most of the manufacturers of programmable instruments including Agilent (HP), Tektronix, Keithley, Fluke, and Racal. The power of SCPI is in its consistent, inclusive and dynamic fundamentals. SCPI is designed to be expanded with new defined commands in the future without causing programming problems. As new instruments are introduced, the intent is to maintain program compatibility with existing SCPI instruments. To match the philosophy behind SCPI, a good SCPI implementation should complement its dynamism yet maintaining consistency. This paper summarizes important features of SCPI and what a configurable implementation must do to complement them. It introduces a Regex-XML driven hybrid implantation designed to complement extensibility, portability and maintainability that is fundamental to SCPI and standards alike.

Keywords – SCPI; SCPI Parser; XML engines; XML driven parser

1. Introduction

Commercial computer-controlled test instruments introduced in the 1960s used a wide variety of non-standard, proprietary interfaces and communication protocols[1]. As programmable instruments became more powerful, so the control languages had to become more complex. For a customer building integrated testing systems and control software, the overhead of learning how to control each piece of equipment had become a major problem[1].

In 1975, the Institute of Electrical and Electronic Engineers approved IEEE 488-1975 was revised and updated in following years. These standards defined the roles of instruments and controllers in a measurement system and a structured scheme for communication. It defined some frequently used housekeeping commands explicitly, but each instrument manufacturer was left with the task of naming any other types of command and defining their effect. Also, it generally did not specify which features or commands should be implemented for a particular instrument. Thus, it was possible that two similar instruments could each conform to IEEE 488.2, yet they could have an entirely different command set.[1]

Standard Commands for Programmable Instruments (SCPI) was introduced in 1990[2]. SCPI created a standard which could be common across all manufacturers and models. The goal of Standard Commands for Programmable Instruments (SCPI) is to reduce Automatic Test Equipment (ATE) program development time. SCPI accomplishes this goal by providing a consistent programming environment for instrument control and data usage. Today, SCPI is supported by most of the manufacturers of programmable instruments including Agilent (HP), Tektronix, Keithley, Fluke, and Racal[3].

In 2002-2003, the SCPI Consortium voted to become part of the IVI (Interchangeable Virtual Instruments) Foundation[4].

The SCPI Consortium evolved to standardize the control language used between programmable instruments. The SCPI Consortium meets once a year to consider modifications to the SCPI Standard. The SCPI Standard is free.[3]

2. Need for configurability

The power of SCPI is in the way it maintains consistency across the devices that implement it and yet it is a very dynamic standard. SCPI is designed to be expanded with new defined commands in the future without causing programming problems. As new instruments are introduced, the intent is to maintain program compatibility with existing SCPI instruments.

A typical static implementation of SCPI will suppress its power. To match the philosophy behind SCPI, a good SCPI implementation should complement its dynamism yet maintaining consistency. We briefly summarize important features of SCPI and what a configurable implementation must do to complement them.

1. **Extensibility**: SCPI is designed to be expanded and new versions will introduce new commands. Also, new versions of device could introduce additional features which SCPI already supports. A configurable implementation must provide a simple way to add new commands.

2. **Portability**: A key to consistent programming is the reduction of multiple ways to control similar instrument functions. The philosophy of SCPI is for the same instrument functions to be controlled by the same SCPI commands. This makes it

portable across instruments. A configurable implementation should also be portable.

3. **Reusability and Maintainability**: Good software is reusable and maintainable. SCPI standard fundamentally supports reusability. A nicely written configurable implementation (like XML driven, with a graphical tools to edit it) can also be maintained by non-programmers.

4. **Scalability**: SCPI defines some common command sets for similar classes of instruments[2]. You can choose to support one or more of these classes according to the type of your instrument. A good implementation should support different scenario that different instruments will introduce.

3. Elements of SCPI programming

3.1. SCPI command Tree

A SCPI command is chain of keywords followed by another chain of parameters. E.g. *":HOR:MAIN:SCAL 1E-6"*. The keywords follow a hierarchical order, like a tree (or more appropriately directed graph as explained later). When a colon is between two command keywords, it moves the current path down one level in the command tree. Following features summarize SCPI lexicon

1. **Command abbreviation:** Each command has a long form and an abbreviated short form. Upper and lower case writing can be used, however, only exact long form or short form is allowed. E.g. *:HOR* and *:HORizontal* can be used interchangeably.

2. **Optional Keywords**: A branch in SCPI command tree represents a SCPI command and certain nodes in a branch are optional. E.g. :*IMMEDIATE* in :*HCOPY:IMMEDIATE* is optional and can be omitted. This feature is what makes SCPI command set a directed graph of keywords rather than a tree.

3. **Multiple commands:** You can send multiple commands within a single program message by separating the commands with semicolons. SCPI implementation should keep track of current path, and the next command is only valid if it includes the entire keyword path from the root of the tree. E. g. *:SOURCE:POWER:START 0DBM; STOP 10DBM*. In this case, START and STOP are at the same level under POWER. Alternatively, a ':' at the start of the command resets the path to root, so we could write the same command as *:SOURCE:POWER:START 0DBM; :SOURCE:POWER: STOP 10DBM*. Note that parameters and semicolon do not affect current path.

4. **Context, Traces and Markers**: SCPI Keywords may carry a notion of a context, a trace or a marker in them. In this case the keyword carries a numeric suffix. E.g. *CALCULATE1:SMOOTHING:STATE ON*. The numeric suffix represents a context, a trace or a marker and typically defaults to context, trace or marker if unspecified.

3.2. SCPI parameter formats

Many SCPI commands accept parameters, which is comma separated list that is separated from the path by a whitespace character. SCPI specifies several data formats that are built over logical, numeric, enumeration, string and binary data types. We present a brief summary here

1. 8, 16, 32 and 64 bit Integers which could be range restricted (bound on lower or upper or both sides) or enumeration restricted (only certain values are acceptable).

2. 32 and 64 bit floating points. These are typically followed by a unit of function (e. G. HZ for FREQUENCY which is optional).

3. Enumeration (specific string representations of numeric values).

4. Free form strings (possibly length restricted).

5. Binary (blob).

6. Arrays, Structures and Unions: Additionally, several commands accept a range of overloaded parameter list that can be best represented by higher order data structures.

3.3. SCPI overloads

A SCPI parameter may be optional. Also, a SCPI command may be overloaded, or could allow two different types of parameter lists. Typically a SCPI command has two overloads

1. GET: To read out a functional value. This is carried out by supplying a question mark *(?)* in place of parameter list.

2. SET: To set a functional value. This is carried out by supplying parameter values.

4. XML representation of SCPI structure

The SCPI representation we propose is hybrid. It has XML base to explain parameter list for a command, format of various parameters, and command behavior. However, the lexicon of command path is expressed using regular expressions.

4.1. Representing command

A <Command> element defines complete behavior of a command. It is specified as

```
<Command
Code="[:SENSe#1]:FREQuency:CENTer">
        <Syntaxes>
                <!—One or more syntaxes -->
        </Syntaxes>
        <Descriptions>
                <!—Descriptions for documentation-->
        </Descriptions>
</Command>
```

Command path is identified by its code. A detailed parameter list and command behavior specification is explained in the syntax node.

4.2. Representing command path

We write command path almost exactly in the same way as it appears in the programmer's manual.

1. Command code consists of tokens separated by `:'. Abbreviated part of keyword is capitalized and rest is in lowercase.

2. An optional keyword is surrounded by square brackets. And

3. An optional context (or trace/marker) specification is represented by a numeric index preceded by #. The numeric index corresponds to an index in Argument list explained later.

Following these rules (and rules for basic data types), lexical analyzer could and has been generated automatically from XML representation.[5]

4.3. Representing syntax

Particular usages of a command are defined by syntaxes (or overloads). Syntax can be defined by command code, role and arguments. Two different syntaxes differ either in command path or in role or in argument list. We specify Syntax as

```
<Syntax Behavior="SYNC" Role="SET">
        <Arguments>
                <!—Zero or more arguments -->
        </Arguments>
        <Returns>
                <!—Zero or more returns -->
        </Returns>
        <Messages>
                <!—Zero or more messages -->
        </Messages>
</Syntax>
```

4.4. Arguments and Returns

<Arguments> encloses a list <Argument> elements which are passed to SCPI parser along with command codes. With <Argument> a format or composite format is always associated which dictates values parser will accept for the argument. We specify an argument as

```
<Argument Name="center frequency" Index="2">
        <Format Code="FREQUENCY" />
</Argument>
```

Attributes `Name' is used for documentation purpose only. The Index should be exactly same as position of the <Argument> element in list under the parent <Arguments> element. Format indicates format of the argument and must be defined previously in `Formats' list. Syntax is validated against every argument in <Arguments> list.

Syntax may have zero or more return value, which are represented by <Return> elements under <Returns>. Returns follow same syntax as arguments.

4.5. Representing Formats

<Arguments> encloses a list <Argument> elements which are passed to SCPI parser along with command codes. With <Argument> a format or composite format is always associated.

5. XML representation of command behavior

A typical command behavior could be explained by a sequence of messages to a proprietary communication library. The format of message could differ from manufacturer to another or even within different versions of software on same device. We present what we think is fair representation of necessary information that would cover a large number of cases.

5.1. Messages

Messages instruct parser what to do when a command is validated. There are more possibilities than just sending a message to or receiving it back from measurement server. A message works in combination with child "Instructions" and "Blocks" element discussed later.

A message is specified as

```
<Message Index="1" Target="PARSER"
Type="GET" Direction="GO" Code="CF">
        <!—Zero or more instructions -->
        <!—Zero or more blocks -->
<Message>
```

The Target attribute indicates who the message is targeted to.

1. **MS Messages (for measurements):** A MS message with `Direction' attribute set to `GO' is sent to MS along with the Blocks attached. A MS message with `Direction'

attribute set to `BACK' is waited for from MS. A message direction could be 'EVENT' representing various events to be processed.

2. **Parser Messages (for parser environment):** Parser messages could be used for housekeeping device, like clearing, setting registers etc. Parser messages could also be used to maintain parser environment (like to maintain session variables for 'session-ful' parser).

5.2. Blocks

A message may contain a list of associated blocks under a child <Blocks> element as <Block> elements. A typical block is specified as

<Block Index="1" xsi:type="BlockArgClass" Datatype="INT 8" Code="REGMASK" Target="ARGUMENT" Value="1">
</Block>

The "Target" attribute specifies what provides or retrieves value of the block. The target can be

1. ARGUMENT/RETURN: The value is obtained from a validated argument or return whose index is specified in the following `Value' attribute, except when Code of the block is "UNIT" the value is obtained from unit for the argument.
2. VARIABLE: Value can come from a session variable, whose name is specified in following Value attribute.
3. VALUE: A string representation of value which can be converted to the blocks data type.

5.3. Instructions

<Message> elements can have a special <Instruction> element attached to them. In this case the instruction is validated before message is processed and message is processed only instruction is valid. For e. g.

<Instruction xsi:type="EQUAL">
<Field Index="1" Code="LHS" Datatype="INT 8" Target="VARIABLE" Value="M LIMTYP/>
<Field Index="2" Code="RHS" Datatype="INT 8" Target="VALUE" Value="LT TIME"
/>
</Instruction>

validates equality of specified fields. The value of the fields is obtained in the same fashion as blocks (with combinations of Target and Value attributes).

6. Optimizations

1. **Automatically generated lexical analyzer:** A lexical analyzer could be generated automatically from the specification.

2. **Binary storage:** As the device environment is constrained, it is best to write a pre-translator program that would translate XML representation into a binary file, which can be transferred to device to be loaded at the startup.

3. **String elimination:** During mapping, all the string keywords could be replaced by a unique numeric hash-code. Lexical analyzer and interpreter should use same hash algorithm.

7. Conclusions

This essay presents a powerful, configurable implementation of SCPI standard that complements the philosophy behind SCPI. The proposal uses structured text formats like regular expressions and XML.

These formats could be maintained easily with help of XML editors or custom graphical tools. The relevant XML schema is free and can be obtained by emailing the author. Author has also developed a successful implementation[5]

8. Acknowledgements

I would like to thank my colleagues who helped me with my work and in preparing report. I also extend my sincere thanks to the head, faculty and staff members of Department of Computer Engineering and Information Technology at Marathwada Mitra Mandal's Institute of Technology, Pune for helping us in various aspects. Last but not least we are grateful to our parents for all their support and encouragement.

References

[1] SCPI Consortium. *Standard Commands for Programmable Instruments (SCPI) Volume 1: Syntax and Style VERSION 1999.0*. SCPI Consortium. May 1999. http://www.ivifoundation.org/docs/SCPI-99.PDF, pp. 1-2

[2] National Instruments. *History of GPIB*. National Instruments. Jun 15, 2009. http://zone.ni.com/devzone/cda/tut/p/id/3419.

[3] JPA Consulting. *SCPI Explained*. JPA Consulting,1999. http://www.jpacsoft.com/scpi_explained.htm.

[4] IVI Foundation. *Integration into IVI foundation.* IVI Foundation. http://www.ivifoundation.org/scpi/default.aspx.

[5] Deshmane, Vikas. *Parser and interpreter for spectrum analyzer*. Final year thesis. Pune University, India. 2012.