THE MODBUS DEFINITION LANGUAGE SPECIFICATION: A FIRST STEP TOWARDS DEVICE INTEROPERABILITY

Jibonananda Sanyal, Joshua New, James Nutaro, David Fugate, Teja Kuruganti
Building Technologies Research and Integration Center
Oak Ridge National Laboratory, Oak Ridge, TN - 37831, USA

ABSTRACT

This paper describes the eXtensible Markup Language (XML) Schema Definition (XSD) for interoperability of Modbus devices. This system uses a common set of XML elements to describe Modbus input and output functions. The goal of the developed technology is to facilitate the rapid, cost-effective retrofit integration of building automation systems by exposing the functions of sensors, actuators, and other data sources through a uniform software interface. Such interoperability facilitates close monitoring of building performance and software-in-the-loop simulation of building energy consumption. The XML Schema is discussed in detail. In addition, two key software utilities, a comma-separated value (CSV) file parser and a device driver generator, are discussed. These helper utilities significantly reduce the barrier to commercial deployment of interoperable Modbus devices.

INTRODUCTION

A key challenge in retrofitting small and medium commercial buildings is the integration of legacy sensors, actuators, and other automation devices with new, increasingly integrated, whole-building control solutions. Integrating legacy assets into these modern architectures can be costly if the legacy system comprises more than a handful of devices. Well-integrated sensors play a crucial role in supplying sensor data to simulation models for calibration purposes and for testing software in retrofit solutions. The integration cost has two parts: discovering what devices and interfaces are available for use, and building the custom software needed to connect existing devices within the integration framework.

Advanced control of HVAC units alone has the potential to reduce whole-building energy consumption by up to 10% (0.5-1.7 of the 17 quads of energy consumed by US commercial buildings) (Bengea 2012, TIAX 2005, Wang et al., 2012). The potential of energy savings by better simulation driven control of other building equipment is also significant. By making retrofits cost-effective for building owners, the proposed technology will accelerate the transformation of these potential energy savings into actual energy savings.

A key challenge in retrofitting small and medium commercial buildings is the integration of legacy sensors, actuators, and other automation devices with new control solutions. The problem of generating device drivers that bind legacy devices to an integration architecture is a less often cited but nonetheless significant driver of cost in retrofitting buildings with new controls. This cost is due to the recurring need for new device drivers to bind new devices to the automation system. Because of the large variety of devices on the market, their numerous variations, and their constantly expanding numbers, a steady stream of new device drivers is necessary to make any integration solution practical and cost-effective.

The goal of the developed technology is to facilitate rapid, cost-effective retrofit integration of building automation systems by exposing the functions of sensors, actuators, and other data sources through a uniform software interface. This will reduce costs by eliminating custom software for integrating legacy devices and by enabling more comprehensive testing of control software by facilitating the creation of modelled devices for software-in-the-loop simulations.

We present a Modbus Definition Language (MDL) specification that defines a standard representation for Modbus device register maps. This allows automated generation of device driver software for Modbus devices, thereby reducing the cost associated with building software to integrate these devices into an integration framework. The MDL is defined by an eXtensible Markup Language (XML) Schema Definition (XSD) that can replace the human-

oriented documentation presently supplied by device vendors with a standardized documentation format suitable for automatic processing. This paper describes the proposed schema for describing a device and how device driver code is automatically generated from an MDL file to enable programmatic communication with the device.

The technology described in this paper is the first major step toward significantly reducing the cost of retrofitting small and medium commercial buildings with advanced controls by allowing equipment vendors to inexpensively and retroactively provide information required for rapid device driver generation for customers. There are also other efforts on data dictionary and data models in building systems that can be leveraged in integrated tool development (BEDES, Haystack).

The rest of the paper is organized as follows: we first describe work related to interoperability facilitating software-in-the-loop simulation; Next we describe the seamless interoperability framework; provide a full specification of the Modbus Definition Language; define the helper utilities that convert current business practices to the new specification; and conclude with a summary and future considerations.

SIMULATION FOR TESTING

Software-in-the-loop simulation is a technique for testing software systems that interact with physical equipment. This testing technique can significantly reduce the cost of testing while simultaneously increasing test coverage (U.S. Congress, 1995). This happens in two ways.

First, software-in-the-loop simulation enables comprehensive testing early in the software lifecycle, before hardware is typically available. Second, by enabling testing activity to take place against simulated equipment, which can be duplicated as needed, it is possible to run more tests than would be possible using relatively scarce hardware resources.

A central part of software-in-the-loop testing is the interfaces to simulated equipment, which must be indistinguishable from the real equipment to the software under test. If every such interface is unique, then the creation and maintenance of models for equipment grows in proportion to the number of devices that must be considered even when their logical functionality is identical; for instance, as would be the case if the control software needed to interact with two different temperature sensors.

By standardizing interfaces to logically identical functions, the cost of building simulated environments for testing is reduced in proportion to reduction in the number of interfaces. The proposed MDL addresses this problem by making it feasible for a standard interface description to be retroactively applied to legacy equipment.

SEAMLESS INTEROPERABILITY

An automated device interoperability framework functions by providing a discoverable, common set of expressive conventions that allow one to communicate with available devices. An integration architecture coordinates communication between the supervisory control and individual device interfaces. An important, practical piece of the developed framework is automatic generation of device drivers from a device description expressed in the MDL format.

The framework specification facilitates the automatic generation of device drivers and capability information that may be retrieved by a device discovery service and/or device description template. When information is incomplete regarding the capabilities of a Modbus device, the skeleton of a device driver is generated to enable partial functionality for the device as well as reduce the labor required to produce a complete device driver.

Figure 1 illustrates the interoperability framework that allows device-discovery and a common device interface for seamless communication with multiple devices. A top-level supervisory control layer allows execution of whole-building control algorithms that can use simulation as well as real-time input to control the building performance. The integration architecture is middleware that exposes a common device interface as well the communication mechanisms for individual device control.

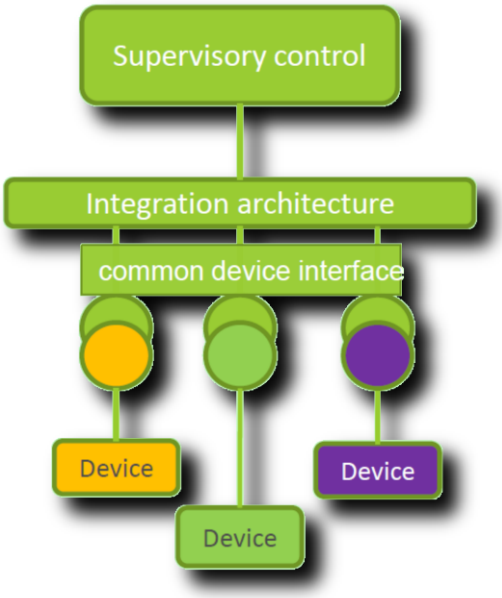

Figure 1: Seamless interoperability framework.

MODBUS XML SCHEMA DEFINITION

The Modbus XML scheme is expressed in the form of a standard XML Schema Definition (XSD) (Bray et al, 1998). An XSD is a recommendation of the World Wide Web Consortium (W3C) and specifies how to formally describe the elements in an Extensible Markup Language (XML) document. The

XSD can be used by programmers to encode and validate XML files to ensure that they adhere to the XSD specification.

XSD Organization

At a high level, the XML Schema Definition (XSD) has a single root element representing the Modbus device. It consists of the device name, description, and a set of functions organized as an `xs:group`. The `xs:group` consists of XML elements that represent read, write, or both read-write functionality of the registers.

Modbus device vendors typically supply register tables that list device addresses and their corresponding functionality. The XML schema is designed based on the functionality supported by the device. As such, the developed XSD proposes a functional listing for a device rather than the more conventional table of registers. Expressing a list of functions of the Modbus device allows for a higher level of abstraction than can be used by programs and tools that focus solely on register-level capabilities. The register address and other information to "read from" or "write to" a device are all encapsulated in the `xs:element` representing the corresponding function. The XSD abstracts the data types for each of the constituent elements of functions. This allows for flexibility in enforcing proper XML syntax while enabling greater flexibility in managing changes as this specification evolves to different market requirements.

The following sections describe the different parts of the XML schema in more detail. A TEMCO ModBus device is used as a candidate example (Temco, 2015).

XML Namespaces

The XSD starts by specifying the XML version and the supported encoding format. Currently, only US-ASCII characters are supported. The `targetNamespace` and an XML namespace of `xmlns:mdl` are defined for the rest of the document.

```
<?xml version="1.0" encoding="US-ASCII"?>
<xs:schema
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
       elementFormDefault="qualified"
   targetNamespace="http://www.ornl.gov/
       ModbusXMLSchema"
   xmlns:mdl="http://www.ornl.gov/
       ModbusXMLSchema"
>
```

Figure 2: Namespace elements used for XML validation

Device Definition

The namespace elements are followed by the root element, which is the anchor point for the entire schema and corresponds to the Modbus device being described. The root element consists of a `name` element, a `description` element, and refers to an `xs:group` named `modbus_functions`. Both the `name` and `description` elements have custom data types which are described later and both are required elements. The `modbus_functions` group may occur at most once in the body of the device element.

```
<!-- definition of device -->
<xs:element name="device">
    <xs:complexType>
        <xs:sequence>
           <xs:element name="name"
                type="mdl:name_type"
                minOccurs="1"
                maxOccurs="1" />
           <xs:element name="description"
                type="mdl:description_type"
                minOccurs="1"
                maxOccurs="1" />
           <xs:group
                ref="mdl:modbus_functions"
                minOccurs="0"
                maxOccurs="1" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Figure 3: The highest-level XML element representing a Modbus device.

Modbus Function Group

The `modbus_function` group is an xs:group which encapsulates a sequence of Modbus function elements. Each element in the sequence is an instance of the data type `mdl:function_type` which describes a function of the Modbus device.

```
<!-- definition of Modbus function group -->

<xs:group name="modbus_functions">
    <xs:sequence>
        <xs:element name="function"
            type="mdl:function_type"
            minOccurs="0"
            maxOccurs="unbounded"
        />
    </xs:sequence>
</xs:group>
```

Figure 4: A group of functions allows Modbus devices to support communications for sensing and control.

Data Types

All elements use a derived data type which allows one to enforce explicit rules on acceptable element values. This design also allows flexibility with respect to future schema changes. The following are the various `xs:element` data types in the schema.

Function Type Definition:

The `function_type` encapsulates a read, write, or a read-write element of the schema and is a fundamental XML element which encapsulates a specific capability of the Modbus device. An xs:sequence of `function_type` instances are used to define the `modbus_functions` entry for a device. The `function_type` is an

`xs:complexType` and constitutes of a sequence of the following elements:

i. Required `name` element of `mdl:name_type`
ii. Required `description` element of `mdl:description_type`
iii. Required list of register addresses of `mdl:address_list_type`. It is an optional element and multiple values must be separated by spaces.
iv. Length element of `mdl:length_enum_type` describing the word length of the register. This is an optional entry and defaults to `Full word`. Other possible values are `Lower byte` or `Upper byte`.
v. Count element of `mdl:count_type` describing the number of `length` elements for the register. It is an optional element and defaults to 1.
vi. Format element of `mdl:format_enum_type` describing the type of native data type of the register. It is an optional element and defaults to `INT8`. Other possible values are `INT16`, `FLOAT16`, and `FLOAT32`.
vii. Block label of `mdl:block_label_type` describing register groups. Some vendors commonly group registers into named categories. It is an optional element that may occur any number of times.
viii. Multiplier element of `mdl:multiplier_type` which is used by the code generator as a scaling factor in the absence of a verbose read or write function description. It is optional and defaults to a value of `1.0`.
ix. Units element of `mdl:units_type` describing the unit of measure, if applicable. It is an optional element.
x. Read capability of `mdl::read_function_type`describing the read functionality used by the code generator to create the device driver. It is an optional element.
xi. Write capability of `mdl::write_function_type`describing the write functionality used by the code generator to create the device driver. It is an optional element.

The optional read and write function types are `xs:simpleType` elements which allows only a string value. Ideally, this is the C code that must be embodied and used in the driver generation. This fragment must be syntactically complete except for the definition of the register variables and the return argument variable. Those are defined by the format elements occuring previously.

Figures 6 and 7 illustrate an example of the `mdl::read_function_type`. In this example, a fragment of C code converts the register values into a data item that is useful to an application. The register variables are of type `uint16_t` and are named `r1, r2, r3`, etc. The return argument variable has the name `arg`. The code divides the register number 1value by 10 and returns the result as a floating point number.

```
<!-- definition of a function type -->
<xs:complexType  name="function_type">
    <xs:annotation>
        <xs:documentation xml:lang="en">
        <p>This element contains the
         description(s) of data item(s) by
         functionality of the device.</p>
        </xs:documentation>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="name"
          type="mdl:name_type"
          minOccurs="1"  maxOccurs="1"/>
        <xs:element name="description"
          type="mdl:description_type"
          minOccurs="1" maxOccurs="1"/>
        <xs:element name="addresses"
          type="mdl:address_list_type"
          minOccurs="1" maxOccurs="1"/>
        <xs:element name="length"
          type="mdl:length_enum_type"
          minOccurs="0" maxOccurs="1"
          default="Full word"/>
        <xs:element name="count"
          type="mdl:count_type minOccurs="0"
          maxOccurs="1" default="1"/>
        <xs:element name="format"
          type="mdl:format_enum_type"
          minOccurs="0" maxOccurs="1"
          default="INT8"/>
        <xs:element name="block_label"
          type="mdl:block_label_type"
          minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="multiplier"
          type="mdl:multiplier_type"
          minOccurs="0" maxOccurs="1"
          default="1.0"/>
        <xs:element name="units"
          type="mdl:units_type"
          minOccurs="0" maxOccurs="1"/>
        <xs:element name="read_function_code"
          type="mdl:read_function_type"
          minOccurs="0"  maxOccurs="1"/>
        <xs:element name="write_function_code"
          type="mdl:write_function_type"
          minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
</xs:complexType >
```

Figure 5: The primary descriptors required for generating drive code for a Modbus device.

SUPPORTING UTILITIES

A device can be described by manually creating an MDL instance which describes the registers and allowable data values for a device; however, this can be tedious and error-prone for device vendors. To streamline the generation of these XML files, software is provided that enables automated

transformation of standard device register map tables in comma separated value (CSV) format to MDL. First, a Python-based CSV parser translates a vendor's device register tables into a prototypical XML file that describes the device and its functionality. Second, a C++ device driver generator creates device drivers from the XML file that allows programmers, or software applications, to easily read from or control each device.

The conversion from device register maps to driver code involves the following 2-step process:

Step 1: Conversion of register maps to MDL-compatible XML files. The register maps, commonly expressed in a tabular manner as a CSV file, can be read into a Python program which creates an XML file adhering to the MDL schema. This is illustrated in Figure 8.

Step 2: Generation of driver code. The MDL-compatible XML file is read by a device driver generator code to create C++ source code for reading and controlling the device. This is illustrated in Figure 9.

```
<xs:simpleType name="read_function_type">
    <xs:restriction base="xs:string"/>
</xs:simpleType>
```

Figure 6: The definiton of read functionality element in the XSD.

```
<read_function_type>
    arg = (float)r1/10.0f;
</read_function_type>
```

Figure 7: An example of a C code snippet that will be used in the device driver generation.

CSV Parser

The CSV parser is a Python script that reads in a device INI-style file that specifies the specific parameters it needs to successfully generate an MDL-compatible XML. The INI file supplies an initial list of functional keywords compiled from multiple devices and vendors. This file is used by the parser to search a register table for keyword matches. As an example, reading and writing the temperature set point for a thermostat device is fairly common. Therefore, the algorithm searches for all instances of 'temperature' in a manufacturer's spreadsheet to determine a possible data item that should be read or written to by a device driver. For exact matches, the script uses values in fields (such as address, length, count, etc.) to create an XML entry for the Modbus function. However, it is impossible to have a completely consistent nomenclature across all device manufacturers. Partial matches are common. In such situations, it calculates a similarity ratio using the Python difflib SequenceMatcher algorithm (Python difflib, 2015) to make a best guess at what each device's functionality is and how it corresponds to known/traditional functionality.

The script may be run interactively or non-interactively. In interactive mode, the script provides a sorted listing of the closest matches and allows the user to determine which field in the manufacturer's device spreadsheet corresponds to the field required by the device's XML data description. In the non-interactive mode, the script uses what it has algorithmically determined to be the closest match.

The device INI file contains meta-information, I/O functionality information, column indices for lookup of required fields, list of synonyms for exposed functionality, and additional user-defined columns for extending programming logic within the script. The list of synonyms is fundamental in automating the matching of functionality to keywords in the process of generating MDL-compatible XML for a given device.

Device Driver Generator

The device driver generator ingests an MDL-compatible XML file. This file is extremely verbose and contains vendor supplied code snippets that show precisely how to utilize the device's functionality. The output of the device driver generator is a <device_name>.h and <device_name>.cc files. This is a fully automated process and requires no interactive human guidance.

CONCLUSION

The paper presents an XML based Modbus Defintion Language and automatic driver generation framework to facilitate rapid, cost-effective retrofit integration of building automation systems by exposing the functions of sensors, actuators, and other data sources through a uniform software interface. This enables new and existing device discovery protocols that can overlay legacy technologies, with varied communication protocols interfaces, and be incorporated into new services which can read and control multiple Modbus devices.

This paves a path twoards building interoperable solutions that can be retrifit in existing building for improving energy efficiciency. The XML schema offers a standardized technique for device enumeration. It is anticipated that the XML schema described here will be used by device vendors to state the functionality of their devices. The expected benefits of adoption are many and range from increased penetration of whole building control systems for inproved building performance to fine-grained individualized control of devices using data-driven approaches.

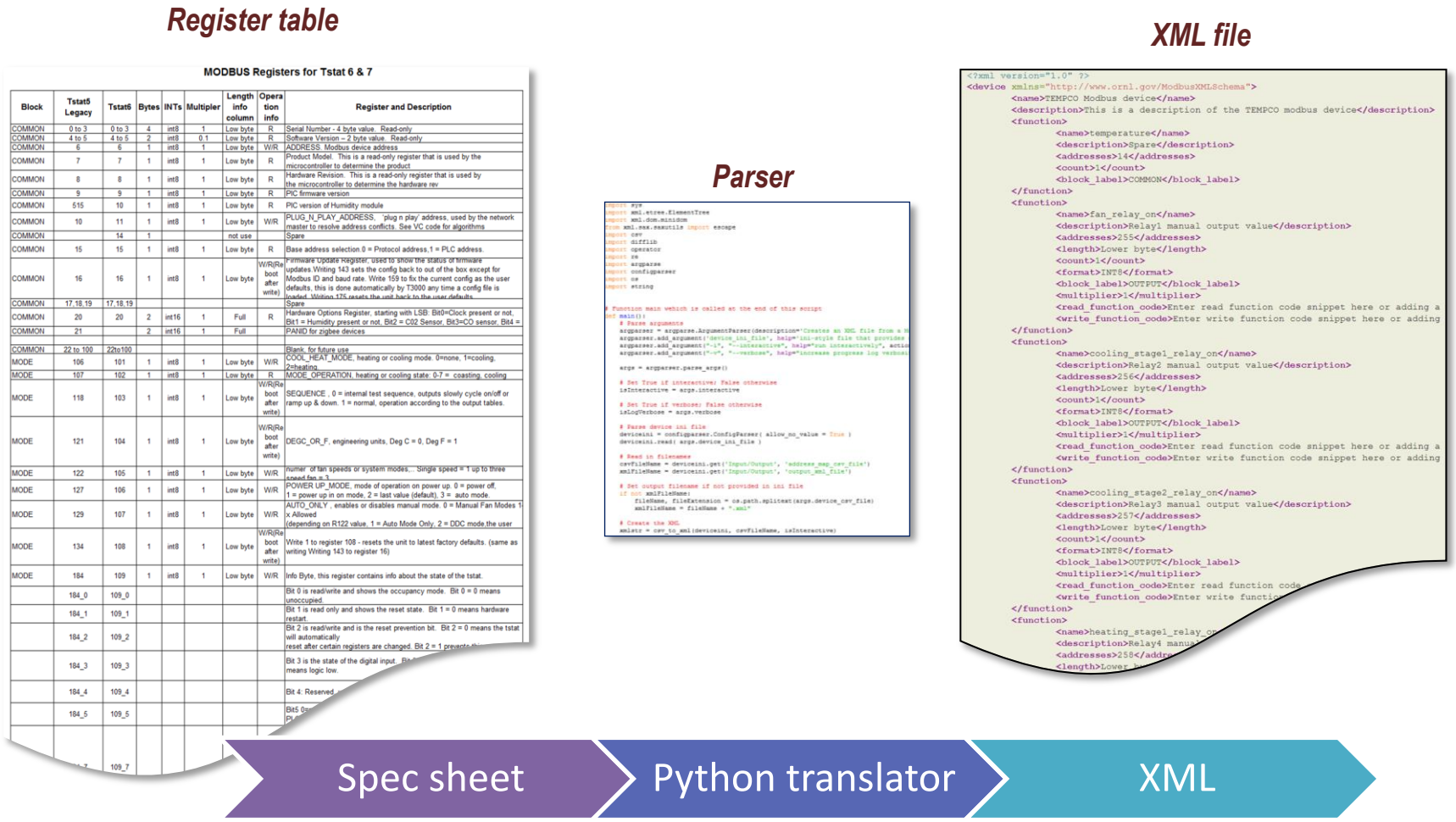

Figure 8: Logical representation of the CSV parser to MDL-compatible XML file

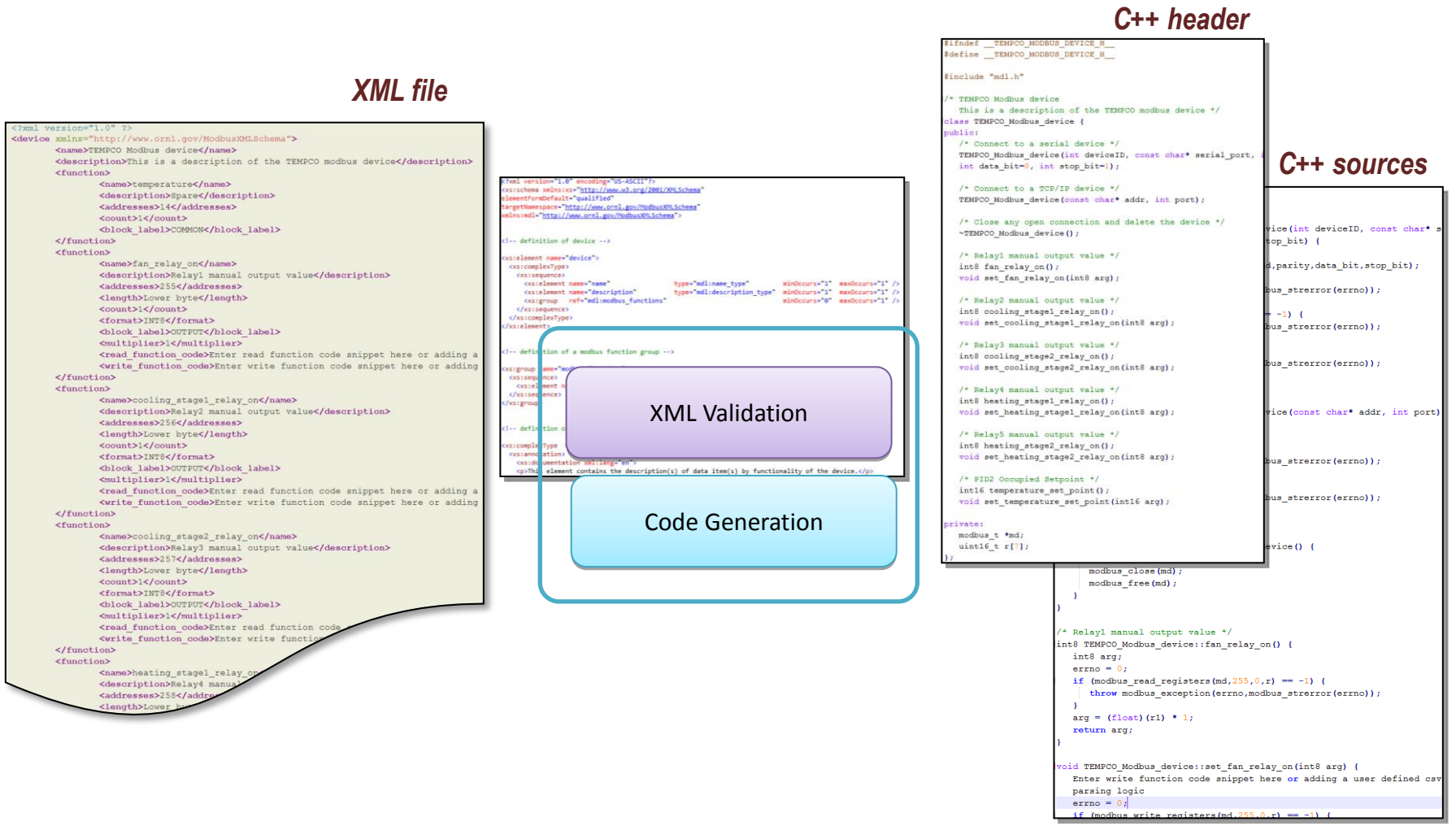

Figure 9: Logical representation of the device driver is validated and then used to generate C++ driver code for communicating with the device.

ACKNOWLEDGEMENT

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). The authors would also like to acknowledge the U.S. Department of Energy, Office of Energy Efficiency and Renewable Energy's Building Technologies Office for funding this work.

REFERENCES

BEDES. Building Energy Data Exchange Specification (BEDES), US DOE, http://energy.gov/eere/buildings/building-energy-data-exchange-specification-bedes, Accessed 14 May 2015.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (1998). Extensible markup language (XML). World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210, 16.

Bryan Gorman and David Resseguie. Final Report: Sensorpedia Phases 1 and 2. SERRI Report 10-89950-1. May 2010.

Haystack, Project Haystack, http://project-haystack.org/, Accessed 14 May 2015.

Liping Wang, Steve Greenberg, John Fiegel, Alma Rubalcava, Shankar Earni, Xiufeng Pang, Rongxin Yin, Spencer Woodworth, Jorge Hernandez-Maldonado, Monitoring-based HVAC commissioning of an existing office building for energy efficiency, Applied Energy, Volume 102, February 2013, Pages 1382-1390.

Python difflib. 2008. Helpers for computing deltas, https://docs.python.org/2/library/difflib.html, Accessed 14 May 2015.

Sorin C. Bengea, Anthony D. Kelman, Francesco Borrelli, Russell Taylor, and Satish Narayanan. "Model Predictive Control for Mid-Size Commercial Building HVAC: Implementation, Results and Energy Savings." The Second International Conference on Building Energy and Environment (COBEE2012), August 1–3, 2012

Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. "sMAP—A Simple Measurement and Actuation Profile for Physical Information." SenSys'10, November 3–5, 2010.

Temco Controls, http://www.temcocontrols.com/, Accessed 14 May 2015.

TIAX 2005. Energy Impact of Commercial Building Controls and Performance Diagnostics: Market Characterization, Energy Impact of Building Faults and the Energy Savings Potential, TIAX Report D0180 for US Department of Energy Contract 030400101, November 2005.

U.S. Congress, Office of Technology Assessment, 1995, Distributed Interactive Simulation of Combat, OTA-BP-ISS-151, Washington, DC: U.S. Government Printing Office.