Automatic Generation of SDM Application Source Code from xTEDS

Jacob Christensen
Utah State University
Logan, Utah, 84321; (435)-797-4704
jacob.h.christensen@aggiemail.usu.edu

Scott Cannon
Utah State University
Logan, Utah, 84321; (435)-797-2015
scott.cannon@usu.edu

Bryan Hansen
Space Dynamics Laboratory
North Logan, Utah, 84341; (435)-797-4392
bryan.hansen@sdl.usu.edu

Jim Lyke
Space Electronics Branch, AFRL / RVSE
Albuquerque, New Mexico, 87117; (505)-846-5812
james.lyke@kirtland.af.mil

ABSTRACT

The Satellite Data Model (SDM) developed at Utah State University (USU) is a plug-and-play software system for satellites. An xTEDS must be written for each component that participates with the SDM. Software applications can query the SDM for data products, commands, and services that components provide. The application can then subscribe, command, and use services from a devices that it has never seen before. The extra functionality required of an SDM application can be difficult to develop. This paper presents a technique for automatically generating SDM application source code by extrapolating information from xTEDS. This technique has been implemented in a tool called the SDM App Wizard. The SDM App Wizard reduces the time and effort required to develop a software application that participates with the SDM.

INTRODUCTION

The Satellite Data Model (SDM)[1, 2] developed at Utah State University (USU) is a plug-and-play software system for satellites. This system allows devices to be connected to the satellite network for self-discovery and auto-configuration and allows applications to use these devices without prior knowledge of the specific device interface. This process is accomplished by the new sensor device providing a description to the SDM of all the data products, accepted commands, and services it can provide. The SDM catalogs this information and makes it searchable by other devices and applications.

Device self description is accomplished through an extensible Electronic Transducer Data Sheet (xTEDS).

The xTEDS is an extension of the IEEE 1451.4 (TEDS) standard, represented in XML. It contains all the information needed to discover and configure a new sensor on the satellite network. The xTEDS is typically stored on an Appliqué Sensor Interface Module or ASIM[3]. When a new device is first connected, it sends its xTEDS to the SDM to be cataloged. Once the device has successfully registered, other devices and applications can query the SDM for data, commands, and that meet application needs.

In order to participate with the SDM, software applications require extra functionality. First, it must be capable of communicating with the SDM, registered devices, and other SDM applications. This level of messaging is done using the Space Plug-and-Play Avionics (SPA) messaging standards[4]. Second, it needs to be able to find, select, and consume data from registered devices. Finally, application can also register with the SDM by providing an xTEDS containing a description of data products, commands, and services that it is capable of providing and accepting. These

extra capabilities represent a significant amount of work. In order to reduce this extra work we have developed a technique for automatically generating much of the SDM application's source code by extrapolating information from xTEDS and structured user input. The rest of this paper follows as outlined. First, xTEDS and the information they contain are discussed. Second, the architecture of a SDM application is covered. Third, a novel technique for automatically generating SDM application source code is presented. Fourth, this technique is demonstrated by presenting the SDM App Wizard. Finally, conclusions and future work are discussed.

XTEDS

Each sensor devices that participates with the SDM is required to provide an xTEDS. An xTEDS describes the data types and messages that a sensor device is capable of producing and accepting. An xTEDS is divided into logical interfaces with each interface describing the data products and messages that it is capable of producing and accepting. Data products are defined with a variable tag and messages are defined with the <Notification>, <Command>, and <Request> tags. An example xTEDS is shown in Figure 1.

The <Interface> Tag

The <Interface> tag is very simple, but very important. The <Interface> tag has three attributes: name, ID, and description. The name attribute is used to give a human readable name to the interface. The ID attribute is used to give a machine readable name to the interface. The description attribute is used to provide a human readable description of the interface. The SDM uses the interface ID and the message ID to create a unique ID for each message inside the <Interface> tag.

The <Variable> Tag

Key parts of each xTEDS interface are the <Variable> tags. The <Variable> tag has three required attributes. The name attribute is used to refer to a variable throughout the rest of the interface. The kind is used for SDM queries. The format declares the data type of the variable. Length is an optional attribute that can be defined when using arrays and strings. Each variable in an interface is only declared once using the <Variable> tag and then referenced by each of the messages that use it. This reference is done with the <VariableRef> tag.

The <Notification> Tag

The notification message is used for data product subscriptions. This makes it the most commonly used message type. The <Notification> tag has a nested <DataMsg> tag that has five attributes. The attributes are: name, description, ID, msgArrival, and msgRate.

```
1 <?xml version="1.0" encoding="utf-8" ?>
 2 <xTEDS name="ExampleDevice_xTEDS" version="1.0">
 3    <Device name="ExampleDevice" kind="tempSensor" />
 4    <Interface name="ExampleInterface" id="1" description="An example" >
 5       <Variable name="celcius" kind="temperature" format="UINT16" />
 6       <Notification>
 7          <DataMsg name="GetTemperature" id="2" msgArrival="PERIODIC" msgRate="1.00">
 8             <VariableRef name="celcius" />
 9          </DataMsg>
10       </Notification>
11       <Command>
12          <CommandMsg name="SetTemp" id="1" >
13             <VariableRef name="celcius" />
14          </CommandMsg>
15       </Command>
16       <Request>
17          <CommandMsg name="GetTemp" description="" id="3" >
18             <VariableRef name="celcius" />
19          </CommandMsg>
20          <DataReplyMsg name="TempData" id="4" >
21             <VariableRef name="celcius" />
22          </DataReplyMsg>
23       </Request>
24    </Interface>
25 </xTEDS>
```

Figure 1 - Example xTEDS

The name and description attributes are for human readability. The ID is combined with the interface ID to form a unique ID for the message. The msgArrival attribute is used to specify the type of arrival for the message. The options are either "PERIODIC" or "EVENT". A periodic notification message is generated at some time interval, whereas the event notification message is generated when a user defined event takes place. Inside of the <DataMsg> tag, there are one or more <VariableRef> tags. These tags refer to the data products that are contained in the notification message.

The <Command> Tag

The command message is used to describe the commands that the sensor device is capable of handling. There is no reply to a command message. The <Command> tag contains a nested <CommandMsg> tag which has three attributes. The attributes are: name, description, and ID. The name and description attributes are for human readability. The ID is combined with the interface ID to form a unique ID for the message. Inside the <CommandMsg> tag, there are zero or more <VariableRef> tags. These tags refer to the data that is sent in the command message.

The <Request> Tag

The request message is like the command message with the exception that there is an expected reply to the request message. A request message contains a command message and a data reply message. The command message is identical to the command message described above. The data reply message is defined with a <DataReplyMsg> tag that is identical to the notification message's <DataMsg> described above. The request message can be used to provide responses to commands or to provide services that contain data products.

SDM APPLICATION ARCHITECTURE

The architecture of an application that participates with the SDM follows a common pattern used by many distributed applications (Figure 2). A listener thread is created to listen for incoming messages. After a message is received it is placed in a queue. The main thread processes messages out of the queue as they become available. This architecture is very responsive and doesn't drop messages because of the dedicated listener thread.

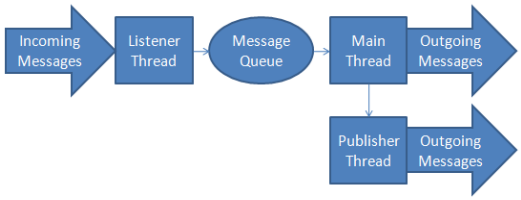

Figure 2 - SDM Application Architecture

The basic tasks that a SDM application must perform are divided among the different components of the application architecture. These tasks are:

- Sending messages
- SDM registration
- Listen and queue incoming messages
- Query, select and subscribe to data
- Dataflow
- Manage subscriptions
- Algorithm and data processing
- Handling commands

The SDM provides a shared library that a SDM application can link in during compilation that contains functionality to handle many of the tasks listed above. The distinction between generated code and library code is small but important: library code cannot be changed by the application developer, but generated code can be modified to fit the needs of the application. Library code is already compiled when the application developer gets it and therefore cannot make changes to it, but the generated code is yet to be compiled and can be changed.

SDM APPLICATION CODE GENERATION

Of the eight tasks that a SDM application must perform, three are handled by SDM library code and three can be automatically generated using information contained in xTEDS and user input. The three that are handled by SDM library code are:

- Sending messages
- SDM registration
- Listen and queue incoming messages

The three that are can be automatically generated are:

- Dataflow
- Manage subscriptions
- Query, select and subscribe to data

And the last two that cannot be automatically generated are:

- Algorithm and data processing
- Handling commands

Sending Messages

One of the first capabilities needed by a SDM application is the ability to send SPA messages[4]. The SDM library defines a class for each SPA message. Each SPA message class derives from the SDMMessage base class that implements the send function. This makes sending a SPA message very simple. First, instantiate the appropriate message and populate the fields (including destination), then call the "Send( )" function. This also handles the data marshalling.

Listen and Queue Incoming Messages

The SDM library provides a utility called the `MessageManager`. The `MessageManager` handles the listening and queuing of incoming messages. The `MessageManager` provides two functions, `IsReady()` and `GetMessage(char* buf)` that facilitate receiving messages (Figure 3). The `IsReady()` function returns true if there is a message in the queue, and the `GetMessage(char* buf)` message retrieves the message and copies it into the provided buffer.

SDM Registration

If the application has data products, commands, or services that it provides to other SDM components, then it is required to register its xTEDS with the SDM. The SDM registration protocol consists of several messages going back and forth between the SDM and the application, with the final message containing the xTEDS for the application. This protocol is handled by the SDM library. The only input needed from the software developer is the file system path of the xTEDS file.

Dataflow

After a message has been received by the `MessageManager` it needs to be delivered to the appropriate handler function for the message and the data it contains. This functionality is automatically generated (Figure 3). In this switch statement messages are sent to the appropriate function based on the message type.

```
if(messageManager.IsReady())
{
    switch(messageManager.GetMessage(buf))
    {
        case SDM_ACK:
            printf("SDMAck confirmation received\n");
            helloReply = true;
            break;
        case SDM_Register:
            printf("SDMRegister received\n");
            waitForReg = false;
            break;
        case SDM_ID:
            printf("SDMID received\n");
            waitForID = false;
            idMsg.Unmarshal(buf);
            myCompId = idMsg.destination;
            break;
        case SDM_Subreqst:
            subReqstMsg.Unmarshal(buf);
            subManager.AddSubscription(subReqstMsg);
            break;
        case SDM_Deletesub:
            deleteSubMsg.Unmarshal(buf);
            subManager.RemoveSubscription(deleteSubMsg);
            break;
        case SDM_Command:
            printf("SDMCommand msg received\n");
            CommandHandler(buf);
            break;
        case SDM_RegInfo:
            RegInfoHandler(buf);
            break;
        case SDM_Data:
            DataHandler(buf);
            break;

        default:
            printf("Bad message received\n");
            break;
    }
}
else
{
    usleep(10000);
}
```

Figure 3 - Dataflow Statement

Manage Subscriptions

Applications that are capable of providing data products have to track subscriptions and publish the data. In Figure 3 cases for subscriptions and cancelations have been generated. The subManager object is an instance of the `SubscriptionManager`. The `SubscriptionManager` is a utility provided by the SDM library to track subscriptions. To handle the publishing of the data products the code in Figure 6 is generated. This code runs in a separate thread that publishes `GetTemperature` message defined on line 7 of the example xTEDS in Figure 1. All of the code to publish this message in accordance with the periodic rate of 1 Hz as defined in the example xTEDS on line 7. This function makes a call to the `Getcelcius1()` function. Because the xTEDS doesn't contain any information about where or how this variable is produced, the `Getcelcius1()`

function is left to be completed by the application developer.

Query, Select and Subscribe

In order to subscribe to a data product, a query must first be issued to the SDM to search for a component that is capable of providing the needed data. The code to execute the query can also be automatically generated with input from the application developer. The required input is the search terms for the query. Using this input the code in Figure 7 is generated. The `ReqRegistrations()` function is used to submit the query to the SDM. The `RegInfoHandler()` function handles the responses from the query. Since it is possible that multiple components can provide the needed data, the application developer is left to complete the section of the `RegInfoHandler()` function that selects to which result to subscribe.

Algorithm and Data Processing

The code needed to handle the data products once they have been received and are ready for processing cannot be generated because the purpose of the application is not captured in the xTEDS. However function stubs are generated so the application developer knows where to place the algorithm and data handling code. The structure of the generated code is such that these functions are called when new data products are ready to be processed.

Command Handling

The code needed to handle a command cannot be generated because the xTEDS does not contain information about how a command should be handled or what it does. However function stubs are generated so the application developer knows where to place the command handling code. The structure of the generated code is such that these stub functions are called when the command is received by the application.

SDM APP WIZARD

The SDM App Wizard (Figure 4) is a web-based application designed to make the discussed generation of SDM Application code as widely available as possible. The application is currently broken down into two major components. Each component tailors to a specific user's needs.

Figure 4 - SDM App Wizard

The first component is the ASIM Tester Application generator which specifically tailors to the needs of an ASIM developer. To create a Tester Application, a developer first must upload his/her device's xTEDS document. A menu-based application is then generated solely from the information contained within the xTEDS. The ability to remotely compile the application from within the web application is also provided as a convenience so that a device developer does not have to worry about setting up cross-compilation environments. This application allows the device developer to test out all of functionality of his/her device on a SPA network without writing a single line of application source code. This includes subscribing/unsubscribing to produced data as well as issuing commands and requests. All data received from the device is parsed and printed to the screen to allow for quick validation and even aid in calibration.

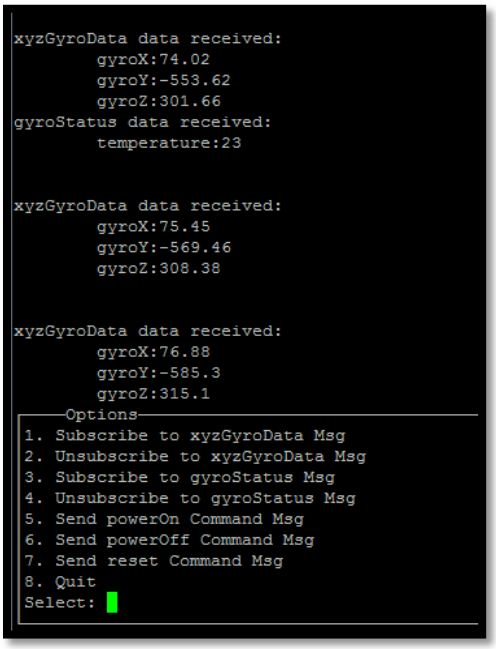

Figure 5 - Tester Application

The second component is the SDM Application Framework generator. This component is tailored towards the needs of the application developer. While the Tester Application was tailored to testing a specific xTEDS, a generated application framework is generated to interoperate with other components in a SPA network of which it does not necessarily need any prior knowledge. If an application provides any data or accepts commands from other components on the network then an xTEDS should be uploaded for the application. All of the code to provide the functionality described in that xTEDS is generated for the user, leaving small sections of code that must be completed by the user as discussed earlier. Users may also complete dialogs within this tool to add queries for data and/or commands to the generated application framework. Again, the majority of the code to issue these queries and handle their associated responses is generated for the user. The code generated by this tool is all ANSI standard C++ and should be useable on any platform.

The SDM App Wizard has been used in several training courses[5] and in several technical demonstrations. As of June 2010, 33 different organizations have participated in training courses using the SDM App Wizard. There have been up to 40 users using the SDM App Wizard at the same time during the training courses. The training course participants provided feedback via a survey at the end of the course. The feedback for the SDM App Wizard was very positive.

Future Work and Conclusion

The current code generation capabilities of the SDM App Wizard have already greatly simplified the somewhat laborious process of writing plug-and-play software, however, there is still much more that can be done. In the near future it is planned to upgrade the SDM App Wizard to produce more logically separated framework code that groups together all code needing completion by the user into a separate file. Future iterations of the tool will also be completely integrated with the Common Data Dictionary (SPA's managed dictionary of searchable terms) to ensure proper queries. Also in the works is allowing users to select a set of xTEDS files that describe other components available on the satellite network. They can then quickly and easily choose needed data products and any commands to issue from those described in the set of xTEDS. Additionally, the idea of fully generating code to exercise a component's functionality can be extended into the automatic generation and real-time reconfiguration of ground station user interfaces based on the multiple xTEDS existing in a system at any given time.

The automatic code generation for SDM source code that has been presented in this paper has helped many climb the learning curve that is required to produce a software application that can participate with the SDM and other components on a SPA network. By reducing the time needed to develop a plug and play application there has been an increased interest in plug and play satellite technologies.

REFERENCES

1. Cannon, S., "Responsive Space Plug & Play with the Satellite Data Model," Proceedings of the AIAA Infotech 2007 Conference, AIAA, 2007.

2. Sundberg, K., Cannon, S., Hospodarsky, T., Fronterhouse, D., and Lyke, J., "A Satellite Data Model for the AFRL Responsive Space Initiative," Proceedings of the 20th annual AIAA/USU Conference on Small Satellites, AIAA, 2006.

3. Christensen, J., Cannon, S., and Hansen, B., "Automatic Software Generation of ASIM Program Code from an xTEDS," Proceedings of the AIAA Infotech 2010 Conference, AIAA, 2010.

4. Lyke, J., "Space-Plug-and-Play Avionics (SPA): A Three-Year Progress Report," Proceedings of the AIAA Infotech 2007Conference, AIAA, 2007.

5. Keif, C. J., Mee, J., Hansen, B., and Christensen, J., "CubeFlow: Training for a New Space Community," Proceedings of the AIAA Infotech 2010 Conference, AIAA, 2010.

```
void* PeriodicDataPublisher(void* args)
{
    double curTime = GetCurTime();

    double GetTemperature_1_lastPubTime = curTime;
    double GetTemperature_1_msgDelay = 1.0/(double)1.00;

    while(!shutdownApp)
    {
        curTime = GetCurTime();

        if(curTime - GetTemperature_1_lastPubTime >= GetTemperature_1_msgDelay)
        {
            char buf_1_2[2];
            unsigned short celcius = Getcelcius1();

            PUT_USHORT(&buf_1_2[0], celcius);
            if(subManager.Publish(GetTemperature_1, buf_1_2, 2))
            {
                GetTemperature_1_lastPubTime = curTime;
            }
        }

        usleep(10000);
    }
    return NULL;
}
```

Figure 6 - Data Publisher

```
void ReqRegistations()
{
    SDMReqReg reqRegMsg;
    reqRegMsg.destination.setPort(myPort);

    reqRegMsg.id = ;
    strcpy(reqRegMsg.device, "someSensor");
    reqRegMsg.reply = SDM_REQREG_CURRENT_FUTURE_AND_CANCELLATIONS;
    reqRegMsg.Send();

}

void RegInfoHandler(char* buf)
{
    SDMRegInfo regInfoMsg;
    SDMConsume consumeMsg;
    MessageManipulator msgManipulator;
    if(regInfoMsg.Unmarshal(buf) == SDM_NO_FURTHER_DATA_PROVIDER)
    {
        //End of stream of RegInfos
        return;
    }
    msgManipulator.setMsgDef(regInfoMsg.msg_def);
    switch(regInfoMsg.id)
    {
        case :
            if(msgManipulator.getType() == NOTIFICATION)
            {
                //TODO: Select which RegInfo you want and consume them as below
                consumeMsg.destination.setPort(myPort);
                consumeMsg.source = regInfoMsg.source;
                consumeMsg.msg_id = regInfoMsg.msg_id;
                consumeMsg.Send();

                strcpy(_msgDef, regInfoMsg.msg_def);
                _compId = regInfoMsg.source;
            }
            break;

        default:
            break;
    }
}
```

Figure 7 - Query, Select, and Subscribe