Development of Distributed Industrial Automation Systems and Debugging Functionality Based on the Open Source OOONEIDA Workbench

Wenbin Dai
The University of Auckland
wenbin_dai@hotmail.com

Arthur Chuan Kang Shih
The University of Auckland
arthur.shih@gmail.com

Valeriy Vyatikin
The University of Auckland
v.vyatkin@auckland.ac.nz

Abstract

The IEC 61499 function block architecture is a new standard proposed to replace the current PLC technologies. Currently the available function block development tools have major shortcoming, specifically the lack of a debug function. The OOONEIDA workbench is an open source function block development tool designed for open development. Our aim is to make the Workbench a fully functioning development tool with emphasis on developing a debug function.

1 Introduction

1.1 Function Blocks and the IEC 61499

New system architecture for distributed automation systems known as IEC 61499 [IEC 2001] is based on the concept of function blocks. A function block is a capsule (e.g. software, but more generally, intellectual property (IP)) that consists of input/output variables, internal variables, and an internal behavior description (Figure 1).

The standard is defined by International Electrotechnical Commission that is an international standard and conformity assessment body for Electrotechnical matters.

In the standard, event drive n function blocks are used as basic blocks for building applications. These function blocks are stored in an XML-based format. This allows for better interoperability between platforms as well as for greater re-usability. There are two types of function blocks, basic function blocks and composite function blocks. Composite function blocks may contain one or more other composite function blocks or basic function blocks. Basic function blocks contains algorithms to control output variables and events, and an Execution Control Chart (ECC) (see Figure 2) to decide which algorithms are called depending on input events.

An application is defined as a network of function blocks.

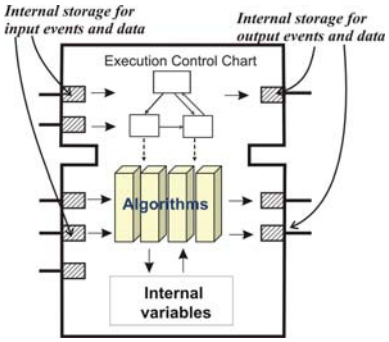

Figure 1: A function block interface [Vyatkin, 2006].

There are two types of function blocks, basic function blocks and composite function blocks. Composite function blocks may contain one or more other composite function blocks or basic function blocks. Basic function blocks contains algorithms to control output variables and events, and an Execution Control Chart (ECC) (see Figure 2) to decide which algorithms are called depending on input events.

An application is defined as a network of function blocks.

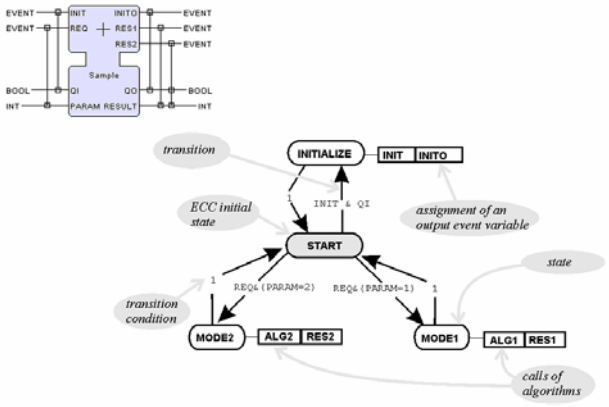

Figure 2: Basic function block [Vyatkin, 2006]

Compiled function blocks can then be combined to form System Configurations that combine the application

logic with device topology, abstract definition of communication networks and exact mapping of function blocks to devices.

1.2 Current Function Block Tools

The current industry leader for IEC 61499 function block development is Function Block Development Kit (FBDK) initially released by Rockwell Automation, and now maintained by Holobloc, Inc. It consists of a development tool (FBDK) and a Runtime (FBRT).

Students taking the Distributed Systems and Computer Networks course at the University of Auckland used FBDK for a major part of their course, and a resulting random survey showed that 90% of the surveyed students wanted a debugging function and 75% wanted a simple way to package function block System Configurations. Further discussion with engineers from Tait Control Systems [TCS, 2006] confirmed the shortcomings of FBDK.

Other function block tools include the CORFU project which includes an UML generator that can generate UML diagrams from function block descriptions, and ISaGraph version 5.0, which is a proprietary commercial product.

2 Workbench Project Scope

2.1 OOONEIDA Workbench

The OOONEIDA Workbench is an open source project started by OOONEIDA and initially developed by Dr. James H. Christensen. It is released under the General Public License.

The main purpose of the Workbench project is to create an open source tool for the development of function block applications in compliance with the IEC 61499 standard. The ultimate goal is to provide an open development platform for all developers similar to what is achieved by the Eclipse™ development environment.
This admittedly can spark the creativity of the researchers using IEC 61499 and help them to capture their experience in form of a versatile and powerful software toolset.

2.2 First Technical Goals of the Workbench Project

The main aim at this stage was to create a debugger for the Workbench. Key functions include displaying the current state of the ECC and input output variables and events. However, in order to do this, workbench must first be able to create, edit, compile and run function blocks. Initially the Workbench was only able to open and view the XML representations of function blocks.

3 Workbench Implementation

3.1 Workbench Compiler

The major existing function block execution environments now are FBRT (Function Block Run-Time by Rockwell Automation) which is embedded in the FBDK (Function Block Development Kit [FBDK, 2006] by Rockwell Automation) and open source project FUBER [FUBER, 2006]. Both of the run-times are implemented in Java and use the Java Virtual Machine to launch the function blocks. FBRT is selected as the run-time for Workbench because it completely supports the IEC-61499 standard, is the mostly wide used and because it is suitable for various environments. For instance, the J2ME version of FBRT is able to run function blocks on embedded devices. The first target for Workbench is to make it compatible with FBRT, however, other runtimes are also envisaged.

FBDK translates the XML representation of function blocks into JAVA source code. The general JAVA compiler is then utilized to compile the JAVA source code to JAVA classes which can be launched by the JAVA Virtual Machine. The following diagram illustrates the normal implementation process of FBDK.

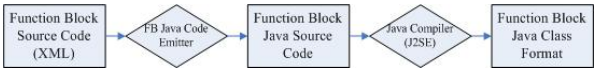

Figure 3. FBRT Compiler Model.

The inconvenience of this method is that after a function block is compiled each time, the user is required to restart FBDK/FBRT to reload the class before it can be used. In the Workbench this problem is solved as follows. Rather than using JAVA code emitter class, Workbench uses eXtensible Stylesheet Language (XSL) and eXtensible Stylesheet Language Transformations (XSLT) to translate XML source code to JAVA source code which can also be compiled by the general JAVA Compiler. XSL is able to translate files stored in XML to a different document type, such as JAVA, by applying an XSLT translation table.

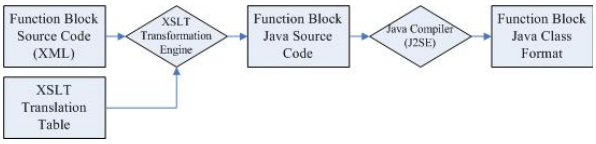

Figure 4. Workbench Compiler Model.

To facilitate the XSL transformation, the XML document converted to a further data format which can be utilized by the XSL translation engine. The JAVA Document extension libraries (DOM) defined by W3C is introduced for building JAVA source code. The DOM library inputs XML files and parses the keywords in the XML to form a tree structure.

The XML translation table is applied then to the new tree structured object. Various templates are built in the translation table. The keywords selected from the DOM Parser will be matched to the template keywords and the corresponding JAVA source code will be generated.

In order to be compatible with the existing run-time, the JAVA source code created by Workbench must appear identical to the code generated by FBRT. The Table shown in figure 5 shows all fundamental templates in the

translation table.

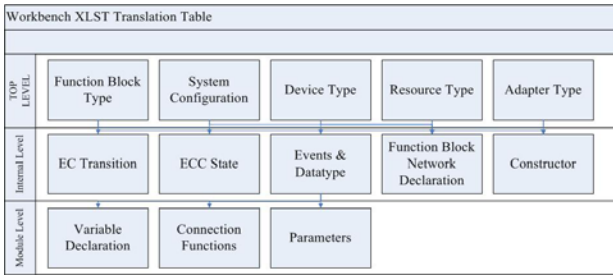

Figure 5. Translation Table Example of XSLT.

The XSL translation is done in three levels as can be seen in Figure 5. The scan process will initially match the keywords in the XML source code of the top level templates. The common type for IEC-61499 objects are function block (including basic, composite and service interface type), Device/Resource/Adapter type and System configuration. When the transformation is processing, the XSLT transfer engine will invoke the internal level templates when the keywords from those levels are found. On the fundamental level, a number of practical functions are implemented in order to be compatible with the FBRT runtime. The translation table will provide identical functionalities as FBDK. A problem rose upon this feature - how to automatically implement sequence of numbers or perform nested loops of JAVA codes in order and check validity of these generated JAVA codes. The semantics of templates and parameters of XSL are applied to this situation as follows: when the variables for counting numbers and increment in sequence or nested loops are required, the translation table will call a sub template internally which passes the variables as parameters (constants here). After the execution is completed, the parameter will be modified and the sub template will be invoked again to perform looping.
As described before, algorithms in the function blocks can be represented in a variety of languages and logics. However, the current version of the Workbench can only support algorithms in JAVA.

3.2 Workbench Launcher and Remote Debugger

The Workbench Launcher is essentially based on the FBRT runtime. The basic execution semantic of FBRT is the call-and-wakeup method. The function block object class will invoke the internal function to call the connected function block which is implements the event generation. When an event is passed to a function block, that function block will be activated and start executing its internal algorithms. These execution processes are controlled and scheduled by the System Manager class in the FBRT package. System Manager supports executing function blocks or system configuration by passing XML source as the input arguments. In addition, several classes in the FBDK tools package are compulsory inserted into the FBRT to facilitate FBRT when parsing the XML to Java source code.

However the new customized FBRT runtime is still not capable for remote debugging functionalities. The structure of new Debug Manager can be seen in Figure 6.

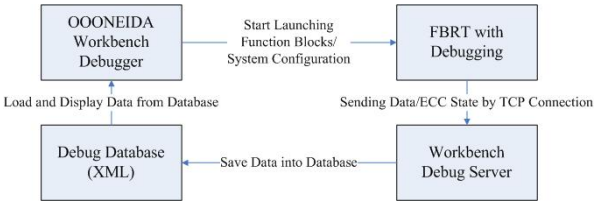

Figure 6. Debug Manager Design.

Workbench will start launching the function block or System Configuration by sending the stored XML files to the FBRT System Manager. The new functions in the new FBRT will start executing the function blocks/system configuration and send the data or ECC state generated by those running instances. When the Debug Server receives these data, the data will be saved into its corresponding position in the database folder in the XML form. Finally, Workbench will reload the data from the debug database for a fixed period and display it to the users.

In FBRT, all function blocks or device types are implemented as class inherited from FBInstance (Function Block basic instance – abstract class). The new debug client class is added into the FBRT in order to create a new TCP connection for sending data. The related sending functions will be integrated into FBInstance class for updating the ECC state, input and output events or data. In the Workbench Compiler, these functions are inserted into the function blocks algorithms which will send data back when any ECC state has changed or Event/Data variables are updated. As the aim of debugging is real-time and remote type devices, the sent data must be in management commands format in XML consistent to the IEC-61499 standard. Management commands are used in establishing connection between remote devices in the function block which enables the systems running distributed over networks. However the IEC-61499, standard does not support any debug keywords in management commands. Figure 7 is an example of the standard format of management commands:

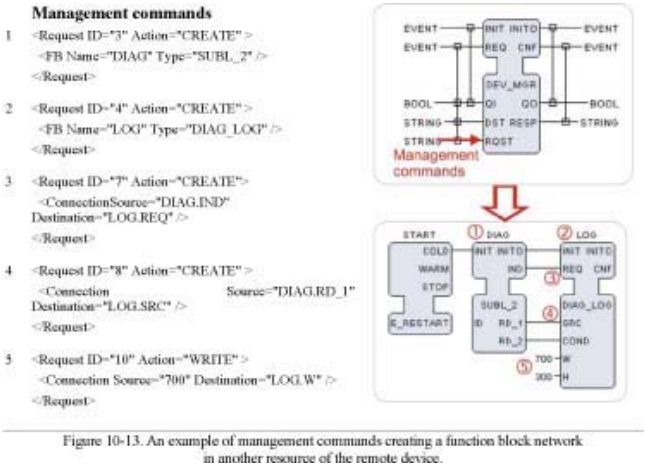

Figure 7. Standard management commands [Vyatkin, 2006].

The added management commands will maintain an identical format to the IEC-61499 standard with newly introduced Action keywords. The new keyword "DEBUG" is used for action type and three special command keywords are defined for updating ECCState (ECC state), DEvent (Events) and Data (Data types). (see Figure 8)

```
<Request ID="1" Action="DEBUG" >
  <ECCState Name="emit1" Type="move_delay_6s.emit_complete" Value="3" />
</Request>
<Request ID="2" Action="DEBUG" >
  <DEvent Name="CNF" Type="move_delay_6s.emit_complete" Value="true" />
</Request>
<Request ID="3" Action="DEBUG" >
  <Data Name="Q0" Type="move_delay_6s.emit_complete" Value="true" />
</Request>
```

Figure. 8. Format of Debug Management Commands

The Parameters are Name, Type and Value which are all compulsory fields. Further details can be viewed in Appendix A: Extension of management commands for remote debugging.

The Debug Server is designed as a multi-thread TCP Server which will listen to the port 61501 (randomly pre-defined). The Debug Server needs to be setup and run externally before launching the function blocks/system configuration. The debug server will return the corresponding response and save data into the debug database in XML form.

The database file is designed to have two sections. The first section is the normal graphical representation of function blocks which is the equivalent to the function block XML representation. The second section will save all the debug management commands received from the embedded devices and keep in database file (fbname.db). Simultaneously another file (fbname.trace) will be updated in a similar manner to the database file, but only the debug management commands of the last received ECC state will be stored. This trace file is designed for graphical view in the Workbench.

The Workbench Debugger will load the trace file of selected function blocks and display it in various views. The debugger currently supports basic function block, device type/composite function block and system configuration debugging. A graphical view of the ECC and interface of input/output events and data will be displayed. For basic function blocks the current state and current updated events/data will be highlighted as well. Function block networks will be illustrated and the last updated events and data will be highlighted for composite function blocks, device type and system configurations. In all cases, the actual value of data variables will be shown in a tabbed panel. There are two updating modes for Workbench Debugger: Manually or Automatic. The manually updating mode will only refresh the graphical representation when the user presses the update button. The automatic mode will refresh itself in a defined period of time.

Place illustrations (figures, drawings, tables, and photographs) throughout the paper at the places where they are first discussed, rather than at the end of the paper. If placed at the bottom or top of a page, illustrations may run across both columns. Securely attach them to the master form with glue stick, spray adhesive, rubber cement, or white tape. Do not use transparent tape as the printing process blurs copy under transparent tape.

Number illustrations sequentially. Use references of the following form: Figure 1, Table 2, etc. Place illustration numbers and captions under illustrations. Leave a margin of 1/4-inch around the area covered by the illustration and caption. Use 9-point type for captions, labels, and other text in illustrations.

Do not use line-printer printouts or screen-dumps for figures---they will be illegible when printed. Avoid screens or pattern fills as they tend to reproduce poorly.

3.3 Workbench Editor

In order for the Workbench to be a fully functional tool for developing Function Blocks, an edit function was required to create and store function blocks. Both graphical and text editor are required.

FBDK uses standard JAVA File Input/Output libraries to edit the XML function block representation files. However, Workbench uses W3C Document Object Model (DOM) objects when manipulating the XML files.

DOM gives more flexibility when manipulating files. When inserting a node using DOM, a simple **insert_node** command is required. Using standard Java Input/Output however requires the whole text document to be read, then the position of the node to be found using string compare commands, then the actual node must then be written to the file before the task is completed. The DOM is a more efficient way of manipulating XML files.

All XML Function Block representations available in the source directories are loaded as DOM objects into a library of DOM objects upon startup of the Workbench. Specific JAVA classes are then used to translate a DOM object into a graphical representation of a function block.

Graphical representations are drawn onto a JLayeredPane. Three separate panes are used, the bottom pane used for drawing the function block shape, including input and outputs; the middle pane used for drawing the interconnections between function blocks; and the top pane is used to highlight selected objects.

A graphical library was written to draw each corresponding element within the IEC 61499 specified XML document. When workbench displays a graphical representation of a function block, the DOM object of that function block is used to read each element of the XML file, and the graphical library is used to draw each element on the JLayeredPane. Figure 9 shows this process.

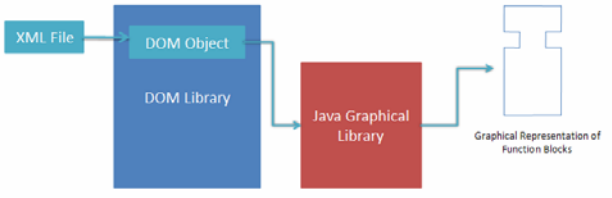

Figure 9. Graphical Representation of Function Blocks in workbench

The text editor implemented in Workbench allows the user to directly change the XML representation of the function block or system configuration being edited. When the file is edited, the document is then parsed back into a DOM object.

In the Workbench text editor, if the XML file is correct, then the object is updated in the original DOM library, while also updating the graphical representation of the function block. If the resulting XML does not satisfy the Document Type Definition (DTD) of the IEEC 61499 specification, then the user will be notified, and the error will also be displayed for the user to fix.

The graphical editor implemented in workbench also changes the original DOM objects and updates the DOM objects library. A mouse listener is implemented in the JLayeredPane that displays a list of editing options, such as "insert new event input" for basic function blocks or "insert new function block" for composite function blocks. When an option is chosen, the parameters required for the option is then required, and once that has been filled in by the user, the information is parsed back into the DOM object, and the DOM objects library is updated.

Workbench also supports editing the internal Execution Control Charts and Algorithms of the function blocks. This can be done in a textual format as well as graphically in the same way as described above. Currently, workbench only supports algorithms writing in JAVA, however with the implementation of an XSLT compiler as explained in section 3.1, there is the possibility of extending this function to support any other language, for example ladder logic or other standard PLC-languages as well as C++ and .NET languages.

3.4 Function Blocks Project Environment

A major concern with FBDK, and other Function Block editors such as CORFU [CORFU, 2006] and ISaGraph v 5.0 [IsaGraph, 2006] is that there is no easy way to move easily System Configurations from one computer to another.

Function Block system configuration consists of networks of function blocks that have already been compiled and can be run automatically. Each System Configuration contains an XML representation of the Function blocks that have been previously compiled. When launched the Runtime will find the location of these compiled classes and run these files as required.

Currently, for a user to move system configurations from one place to another, they have to manually search for each XML representation of the function block, as well as the compiled JAVA classes of the function blocks and package them manually. The user then has to place the packaged file into identical directories on the new computer in order for it to run correctly. This is a very time consuming process, and is very susceptible to errors, such as accidently packaging the wrong file, or forgetting a file which may cause major problems when running automation objects.

The main function of the Function Blocks Project Environment (FBPE) is a simple System Configuration Import Export tool. The FBPE allows the user to choose through all the system configurations located on computer and select which one to export. Figure 10 shows a screenshot of the tool.

When a system configuration is chosen to be exported, it automatically searches through the source folders for the required XML descriptions as well as the compiled JAVA Classes and combines it into a JAR file. A project file is also created detailing where each file is supposed to be kept so that identical directories can be created on the target computer when importing projects.

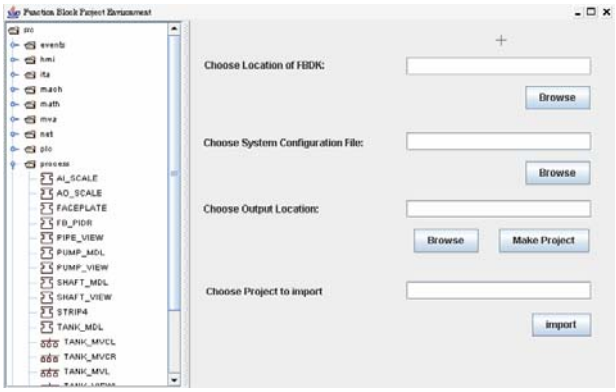

Figure 10. Screenshot of Function Block Project Environment

When choosing to import a project, the FBPE just looks through the project file and replaces each component where it is required, and the system configuration is then successfully ported onto a new computer.

Furthermore, the FBPE also has the added functionality of being a launcher for the Workbench. On the right hand panel there is a scroll panel showing a tree of all the XML Function Block files, and if the user double clicks on one, it will open that function block in the Workbench. This allows for editing of system configurations or function blocks before they are exported into a project file for portability.

The project environment is also designed to be able to check if System Configurations can be used on particular remote devices. The Project Environment will check if the remote devices have the necessary compiled Java classes to run the System Configuration. If not, it will simply upload the missing or updated classes onto the remote device.

4 Further Development of Workbench

The most critical part of Workbench that requires further development is a new execution environment. Currently, Workbench uses a slightly modified Function Block Run Time used by FBDK. Further development of this project must include developing a new execution environment so that Workbench may run independently of FBRT and become a stand alone development tool. Furthermore we propose the use of FUBER, an open source IEC 61499

execution environment, in the development process.

Another important function to be developed would be support for other languages when writing function block algorithms. Currently support is only available for JAVA algorithms. However, developers may want to implement algorithms in different languages other. The use of an XSLT compiler makes this a reasonably straight forward task. Only new translation tables for different languages would be required.

Other developments required include a ladder logic and SQL database interaction. This would allow users to make the transition from PLCs to function blocks easier, and would make function blocks compatible with current industry standards. It will also allow developers to integrate function block applications with their database.

5 Conclusions

From our work, we can conclude that:

- Workbench is now a full functioning, albeit bug-ridden function block development tool with edit, save, compile, and run functions
- Edit function now includes an error checker to check the new XML file conforms with IEC 61499 standard.
- A new debug manager has been successfully created.
- A Function Block Project Environment has been created as a tool for transporting System Configurations from computer to computer, and as a launcher for workbench.

Future work on the project includes:

- Development of an execution environment;
- ECC algorithms in different languages;
- Ladder logic descriptor for function blocks;
- SQL database compatibility.

References

[IEC 2001] Function blocks for industrial-process measurement and control systems - Part 1: Architecture, International Electrotechnical Commission, Geneva, 2005

[Vyatkin, 2006] IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design, 2006, 271 pp., O3NEIDA - ISA

[FBDK, 2006] Holobloc FBDK website: http://www.holobloc.com/doc/fbdk/

[TCS, 2006] TCS website : http://www.rockwellautomation.com.br/applications/gs/ap/gsnz.nsf/pages/Tait_Control_Systems_Limited

[CORFU, 2006] CORFU website : http://seg.ece.upatras.gr/Corfu/

[IsaGraph, 2006] ISaGraph Website: http://www.arcom.com/products/pcp/pcp16.htm

[o3neida, 2006] OOONEIDA: http://www.oooneida.info

[FUBER, 2006] FUBER: http://sourceforge.net/projects/fuber

Appendix A

Extension of Management Commands for Remote Debugging

A.1 Debug Management Command for ECC State:

<ECCState /> is indicated for ECC state changes.

Example:

```
<Request ID="1" Action="DEBUG" >
  <ECCState Name="emit1" Type="move_delay_6s.emit_complete" Value="3" />
</Request>
```

ID	Integer Number which indicates sequence number
Action	"DEBUG" for remote debugging
Name	Current Function Block Name
Type	Hierarchy Function Block Type Name Priority: System Configuration > Device Type > Composite Function Block > Basic Function Block For Example: Move_delay_6s.emit_complete Move_delay_6s = composite function block type name emit_complete = basic function block type name
Value	Value for ECC State, In sequence of Integers from 1. Each Integer Number represent a state

A.2 Debug Management Command for Events:

<DEvent /> is indicated for a new event rose.

Example:

```
<Request ID="2" Action="DEBUG" >
  <DEvent Name="CNF" Type="move_delay_6s.emit_complete" Value="true" />
</Request>
```

ID	Same as stated in ECC Section
Action	Same as stated in ECC Section
Name	Current Input/Output Event Name
Type	Same as stated in ECC Section
Value	Event defined as Boolean type, true equals the event is risen, otherwise equals false

A.3 Debug Management Command for Data Types:

<Data /> is indicated for a data type variable changed its value.

Example:

```
<Request ID="3" Action="DEBUG" >
  <Data Name="QO" Type="move_delay_6s.emit_complete" Value="true" />
</Request>
```

ID	Same as stated in ECC Section
Action	Same as stated in ECC Section
Name	Current Input/Output Data type variable Name
Type	Same as stated in ECC Section
Value	Data Value, String type which represent all types including String, Integer, Boolean

A.4 Debug Management Command Response from Debug Server

Example:

```
<Response ID="1" reason="" />
```

ID refers to the request ID and reason field only occurs ERROR when the data is not correctly stored in the database.

Appendix B: Student Survey Results

Twenty-three students out of approximately sixty students that were enrolled in the Computer Systems 405: Computer Networks were posed with two questions. The first one was "What is your main complaint with FBDK?". The results are shown in table one:

Answer	no. of responses
No Debug Function	20
Can't Zoom in on ECC	2
Too many bugs	1

Table 1: Results for Question 1.

The second question posed was "What would be the two things you would add to FBDK?" The results are shown in table 2 below:

Answer	no. of responses
Debug Function	21
ECC Zoom	4
Project Porting System	17
Other	4

Table 2: Results for Question 2

As can be seen by these results, the two most important things that the students wanted were a Proeject porting system and some sort of debug function.