Implementing a Condensed Scripting Language in the Scripted Battle Management Language Web Service

Dr. J. Mark Pullen, Douglas Corner, Samuel Singapogu,
Bhargava Bulusu, and Mohammad Ababneh
C4I Center
George Mason University
Fairfax, VA 22030, USA
+1 703 993 3682
{mpullen, dcorner, ssingapo, bbulusu, mababneh}@c4i.gmu.edu

Keywords:
Battle Management Language, Web Services, JC3IEDM

The approach to defining a coalition battle management language (BML) now being pursued by SISO requires mapping of BML into a JC3IEDM database, which is accessed via a Web service. In previous SIW papers we have reported on a new approach to implementing such a Web service, based on the notion of an interpreter module. This scripting engine takes as its input the schema of the Web service and a script, coded in XML, that defines the mappings concisely. The Scripted BML Server (SBML), which is available as open source software, has the virtues that it is quicker and easier to change than a hard-coded service and also requires a lower level of expertise for development, once the interpreter has been completed. SBML now includes a publish/subscribe capability, multi-threading for improved performance, and a condensed scripting language for greater ease of script coding. We also have developed a BML C2 GUI with graphical interface and the ability to adapt automatically to BML schema revisions. This paper describes in detail our implementation of the condensed scripting language in SBML.

1. Overview

This paper describes how and why the Scripted Battle Management Web Service (SBML) was extended with a condensed scripting language.

2. SBML Background

Battle Management Language (BML) and its various proposed extensions are intended to facilitate interoperation among command and control (C2) and modeling and simulation (M&S) systems by providing a common, agreed-to format for the exchange of information such as orders and reports. This is accomplished by providing a repository service that the participating systems can use to post and retrieve messages expressed in BML. The service is implemented as middleware, essential to the operation of BML, and can be either centralized or distributed. Recent implementations have focused on use of the Extensible Markup Language (XML) along with Web service (WS) technology, a choice that is consistent with the Network Centric Operations strategy currently being adopted by the US Department of Defense and its coalition allies [1].

Experience to date in development of BML indicates that the language will continue to grow and change. This is likely to be true of both the BML itself and of the underlying database representation used to implement the BML WS. However, it also has become clear that some aspects of BML middleware are likely to remain the same for a considerable time, namely, the XML input structure and the need for the BML WS to store a representation of BML in a well-structured relational database, accessed via the Structured Query Language (SQL). This implies an opportunity for a re-usable system component: a Scripting Engine, driven by a BML Schema and a Mapping File, that accepts BML *push* and *pull* transactions and processes them according to a script (or mapping file, also written in XML). While the scripted approach may have lower performance when compared to hard-coded implementations, it has several advantages:

- new BML constructs can be implemented and tested rapidly
- changes to the data model that underlies the database can be implemented and tested rapidly
- the ability to change the service rapidly reduces cost and facilitates prototyping
- the script provides a concise definition of BML-to-data model mappings that facilitates review and interchange needed for collaboration and standardization

The heart of SBML is a scripting engine, introduced in [2], that implements a BML WS by converting BML data into a database representation and also retrieving from the database and generating BML as output. It could implement any XML-based BML and any SQL-realized underlying data model. Current SBML scripts implement

the Joint Command, Control and Consultation Information Exchange Data Model (JC3IEDM). In the following description, any logically consistent and complete data model could replace JC3IEDM. Reference [3] describes the second generation of SBML.

The current SBML implementation and scripts support two JC3IEDM database interfaces, as shown in Figure 1: one is a direct SQL interface, used with a MySQL database server. The other, SIMCI_RI [4], passes java objects through Red Hat's Hibernate persistence service, which performs the actual database interface function. Version 2 implements a publish/subscribe capability, using the Java Message Service (JMS) as implemented by JBoss in open source (see http://www.jboss.com). Version 2 also implements the XML Path Language (XPATH) (see http://www.w3.org/TR/xpath), wherever a relative path in the XML input is required.

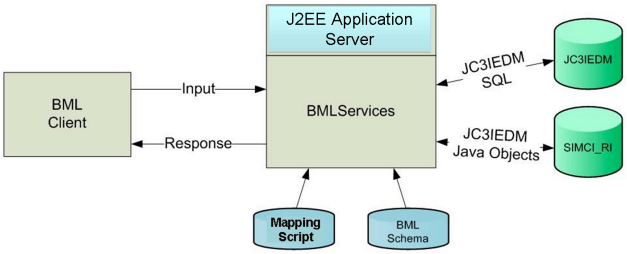

Figure 1: SBML Configuration

The BML/JC3IEDM conversion process is accomplished under control of the scripting language, which is described in [3].

3. Publish/Subscribe in SBML

3.1 Polling Interface

M&S systems generally process two kinds of messages, orders and reports. Orders are larger and more complex than reports, however, the frequency of reports is much higher. For example as a military unit moves from one position to another the M&S system may generate updated position reports every few seconds. With the previous release of the SBML service it was necessary for C2 systems to poll the server to determine if updated report information was available. This is inefficient in several ways:

- The server must reread information that may have just been written to the database
- The poll may retrieve no new information
- The request for each client will be processed separately

Client applications have no way to determine what messages might have been posted to the server since sending the last query; they must first determine what reports are available. Scripting was developed within the SBML framework to retrieve a list of reports posted over a specified period of time. It was then necessary for the client to format a query to retrieve those messages that might be of interest. The precision of this method turns out to be low in that many of the messages retrieved tend to be status reports that are then updated by later messages in the same group. An additional problem is that it is necessary for each client using the server to perform this process in parallel.

The load resulting on the server from polling and the large amount of redundant network traffic clearly is inefficient. In this environment it was necessary for M&S systems to limit the rate of report production so that the server was not overloaded.

3.2 Publish/Subscribe Implementation

A publish subscribe capability was added to the SBML server in order to eliminate the inefficiencies of the polling interface. SBML runs under JBoss 4.2.3 (see http://jboss.org). The messaging service provided by this release of JBoss, JBoss Messaging or JBossMQ, is an implementation Java JMS 1.1 [6]. JBossMQ provides both point to point messaging between two entities (JMS Queues) and a subscription based distribution mechanism (JMS Topics) for publishing messages to multiple subscribers. JMS provides reliable delivery of messages for all subscribers to a particular topic.

SBML version 2.3 provides a set of preconfigured JBossMQ Topics, which are used for the distribution of incoming orders and periodic reports to any interested subscribers. As BML messages are received they are processed by the appropriate script and written to the database. The successful completion of the transaction is an indication that there were no errors in incoming data and that the message can be forwarded to subscribers. There is an XPath [7] statement associated with each Topic, which serves as a filter to determine whether each received message should be written to that Topic. If application of the XPath statement to the message results in non-null result, the message is written to that Topic. A particular BML message may match more than one XPath statement and therefore could be transmitted to more than one Topic. A client then might receive the same message more than once. The publish/subscribe architecture is depicted in Figure 2 below.

Figure 2. Publish/Subscribe Architecture for SBMLJMS is built for the Java environment and uses Java Remote Procedure Calls (RPC). This presents an additional requirement for programs written in C++ (and other

languages) clients. We provide an interface for C++ users, built under the Java Native Interface (JNI) framework. This interface works well, however it does separate the actual client code from a direct interface with the messaging service.

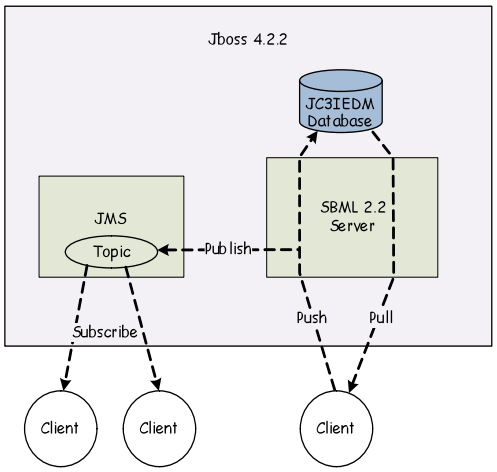

Figure 2. Publish/Subscribe Architecture for SBML

4. SBML in NATO MSG-048

The NATO Modeling and Simulation Group Technical Activity 48 (MSG-048) was chartered in 2006 to investigate the potential of a Coalition Battle Management Language for multinational and NATO interoperation of command and control systems with modeling and simulation systems. The 2009 experimentation phase of MSG-048, described in [5], was performed at George Mason University's Manassas, Virginia Campus in November 2009 and included six C2 Systems and five simulation systems from a total of eight NATO nations, as shown in Figure 3. This was the first large scale use of the SBML Publish/Subscribe service. The service performed well and enabled a much larger set of systems to interact as well as increasing the precision of reports by permitting a higher status update rate. BML messages were batched, to reduce the impact of Web service overhead. The processing time supported by our server for MSG-048 experiments was 50 ms per transaction.

As shown in Figure 2, the publish/subscribe we implemented requires a client module to open a connection to the server, which is used to transmit Web service transactions matching the subscription. The C2 systems and simulations used by MSG-048 in 2009 were implemented in Java and C++. In order to support publish/subscribe under the latter, we provided an interface module made on the Java Native Interface (JNI).

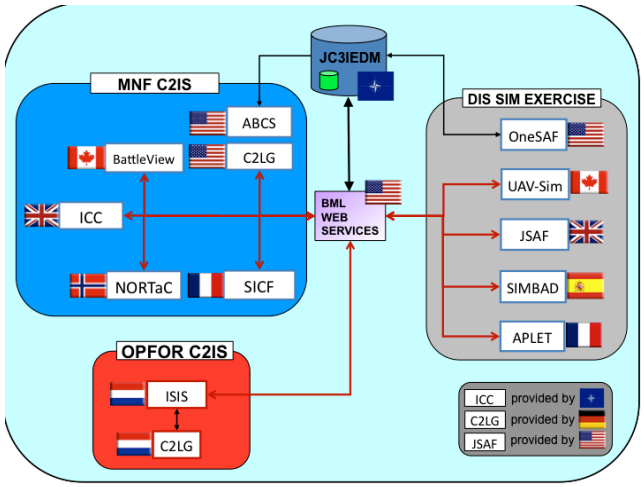

Figure 3. NATO MSG-048 Experimentation Configuration 2009

5. Publish/Subscribe Enhancements

Topics. Under the current implementation, SBML 2.3, the Publish Subscribe service has the topics hard coded along with corresponding XPath statements used for matching messages with JMS Topics. While acceptable in a prototyping environment, this necessitates the rebuilding of the service as requirements change. We therefore plan that, at a minimum, in a future version of SBML the Topics will be configurable through an external file. A more sophisticated enhancement would be a dynamic topic capability such that clients could create a topic in an XML message by specifying the topic name and specifying an XPath statement, used to match against incoming messages. The topic would then be active until the next time the application server was restarted. To prevent name collisions under such a dynamic environment, the server may provide for the publication of new topics are they are created, so that the Topics may be shared between clients.

Client Language. Release 5 of JBoss is in common use and, among other things, provides for a choice of protocols to be used with the messaging subsystem. Upgrading to release 5 of JBoss will allow a more direct interface for C++ clients.

6. BML C2 GUI

The BML client depicted in Figure 1 may be relatively complex to develop, requiring a variety of tools and capabilities such as: an XML editor, operating system command line knowledge, and Java programming experience. The difficulty is greater because the developer doesn't have the opportunity to see the actual geospatial information in the BML documents on a representative

map. Inspired by the way these problems were addressed in Fraunhofer FKIE's C2 Lexical Grammar GUI as described in [5], we have developed the "Battle Management Language Command and Control Graphical User Interface" (BMLC2GUI) as part of the open source tools associated with SBML. The BMLC2GUI is an open-source user interface tool that represents information flowing to/from C2 and simulation systems in text and image formats.

The main purpose of the BMLC2GUI is to provide an easy-to-use graphical user interface to BML users and developers that can serve as a surrogate input/output GUI or alternately to monitor (and if necessary revise) BML documents flowing to/from BML client systems. BMLC2GUI is a Java application, that generates an interface using other open-source tools: Xcentric's JAXFront (see http://www.jaxfront.org), which generates a GUI interface at run time for any XML document, based on its schema; and BBN's OpenMap (see http://openmap.bbn.com), which is used to display geospatial data and control features residing in the BML document (orders or reports) on the map. Figure 4 shows a screen shot of the GUI.

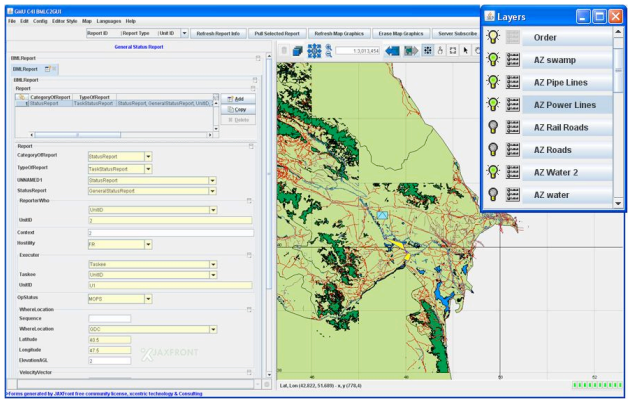

Figure 4. The BMLC2GUI

The BMLC2GUI provides easy and efficient means for the end user to edit, validate, and push BML orders to the SBML Web Services and also to pull and view BML reports from the services. The BMLC2GUI also can subscribe to the SBML subscription service so that the GUI will be updated whenever a new report is pushed by another client under the publish/subscribe capabilities described in sections 3 and 5. The map will depict geospatial information from the BML document the user is editing or revising and display the correct symbols representing the objects or units in addition to all the mapping capabilities intrinsically provided by OpenMap. BMLC2GUI is using the OpenMap implementation of military standard MilStd2525b for unit and object symbol representation [9].

Editing and issuing a BML order or report is very easy using the GUI, requiring only data field entry and selection of items from drop down lists, which are populated automatically from enumerations in the associated schema. The GUI can accommodate changes in BML schemas easily because all of the GUI generation happens at run time. Furthermore, the GUI satisfies the need of experienced users by providing a serialization method of the BML document enabling XML manual edit, validate, save and push capabilities.

We will describe the development and operation of the BMLC2GUI in greater detail, in a paper now under development for SISO SIW Fall 2010.

7. Scripting Enhancements

In the process of developing the scripts described above, we learned some lessons about how to make scripting more practical. Our initial plans in this direction are described in [8]. This paper contains details on the capability as implemented.

7.1 Nested *if-then-else* construct

The logic required in scripts turns out to entail some decisions that are not easy to encode with the single level of if-then-else described in [3]. Therefore, have added a nested *if* to the XML scripting. Our nested ifs mimic the use of "curly braces" { } in progamming languages such as Java. Thus a Business Object (BO) in the script could contain:

```
<BusinessObjectTransaction>
    <transactionName>transName</transactionName>
    [Other BOT header elements]
    <codeBlock>
        <SBML elements>
    </codeBlock>
</BusinessObjectTransaction>
```

If statements within the top level <codeBlock> appear as:

```
<ifThenElse>
    <compare>someWV</compare>
    <relation>EQ</relation>
    <literalValue>xxx</literalValue>
</ifThenElse>
<codeBlock>
    <SBML elements executed if condition was true>
</codeBlock>
<codeBlock>
    <SBML elements executed if condition was false>
</codeBlock>
```

<codeBlock> thus serves as the parent of all language elements. A positive side effect of this is that formatting the script with XML editing tools results in automatic indentation of the elements in each block, making the script much more readable.

7.2 Condensed Scripting Language

The XML-based script used by SBML 2.3, while simple to implement in the Web service environment, is less than optimal for the human programmer since it suffers from the well-known verbosity of XML. We have developed a front-end translator that can reduce the visual and cognitive burden on the script developer by reducing the script to a condensed representation. The Condensed Scripting Language (CSL) is neither more nor less powerful than the XML form, since each can be produced directly from the other by translation. The CSL is intended to be more usable in that it is easier to write. Also, because a logical operation can fit on a single line, comprehending the operation of a CSL script may be easier than for an equivalent XML script. The translator from CSL to XML script and the translator from XML script are available as part of open-source SBML from http://c4i.gmu.edu/BML.

Appendix 1 provides a Backus-Naur Form (BNF) representation of CSL. The whole script is treated as a *BusinessObjectInput* that can contain multiple instances of *BusinessObjectTransaction*. Each of these contains a set of database operations that are intended to leave the Database in a consistent state at the end of the transaction if executed atomically, that is, without interleaving of other BusinessObjectTransactions that operate on the same database tables. Each BusinessObject is identified by a unique identifier and may optionally include a definition of a *multivalueWorkingVariable* and the variable name that it should be assigned to at every iteration of the *BusinessObjectTransaction*. The definition is then followed by a declaration of the parameters and the Return values. The parameters are declared as an ordered list of *variable*s where a *variable* is either a string literal enclosed in quotation marks or a string literal which maps to a *workingVariable* in the XML script. Every *businessObjectTag* is treated as *workingVariable* by the SBML engine. The *BusinessObjectTransaction* thus can contain any number of elements. CSL is like Java in that any amount of "whitespace" (spaces, newlines, and comments) is treated as a delimiter.

CSL offers three ways to retrieve from the database: (1) *GetRow* retrieves a row, (2) *GetList* retrieves a list of elements from a column, and (3) *Get* retrieves a single column entry. In all three commands, the first identifier is the table name, the second is the column name, and the third is a set of *columnReference*s that constitute the *where* clause of the underlying SQL statement. A *Put* operation is defined as a combination of the table name and *columnReferences* that define columns to be updated.

The *Call* statement invokes a *BusinessObjectTransaction* by specifying the name of the BO, the *anchorTag*, and lists of parameters and optional return values. Conditional statements in CSL are *IfThen* and *IfThenElse.* Both of these perform a logical comparison of the identifier with the variable and conditionally execute the statements, depending on the result of the comparison. The *Assign* command is used to assign the value of a variable to a different identifier.

There can be multiple *BOReturn* statements inside a BO and each *BOReturn* can have a unlimited number of the following statements: *HigherTagStart, HigherTagEnd*, and *Tag*. *HigherTagStart* creates a opening tag corresponding to the variable name that is specified; *HigherTagEnd* creates a matching closing tag. The *Tag* statement is used to create a tag with the name specified after the first whitespace and the value specified after the second whitespace. This scheme allows for creation of nested tags and also of tags dynamically named using variables. Figure 5 illustrates a condensed-language script. Appendix 2 contains the XML version of the same script, which is more than four times as long.

```
BOInput
{
  BOTransaction WhatWhenPush(...
  {
    //fragment from WhatWhenPush
    Call TaskeeWhoPush TaskeeWho (task_act_id) () ;
  ...
  }

BOTransaction TaskeeWhoPush (task_act_id) ()
{
   GET unit unit_id (formal_abbrd_name_txt EQ UnitID);
   PUT act_res (
      act_id EQ task_act_id,
      act_res_index EQI act_res_index, cat_code EQ "RI",
      authorising_org_id EQ unit_id) ;
   PUT act_res_item (
      act_id EQ task_act_id,
      act_res_index EQ act_res_index,
      obj_item_id EQ unit_id ) ;
   BOReturn
   {
      BOReturnElement
      {
         Tag Result "OK";
       }
   }
}
```

Figure 5. Condensed script for SBML

7.3 Implementation

To build the Condensed BML generator we used a parser generator: a tool that reads a BNF grammar specification and converts it to a Java program that can recognize matches to the grammar. We used *Java Compiler Compiler* (JavaCC - see https://javacc.dev.java.net), a parser generator for Java applications. In addition to the parser generator itself, JavaCC provides other capabilities related to parser generation such as tree building (via a tool called JJTree), actions, debugging, etc. JavaCC works with any Java VM version 1.2 or greater.

JavaCC itself is not a parser or a lexical analyzer but a *generator* that produces lexical analyzers and parsers coded in Java. It produces them according to a specification that it reads from a file. For example, the rules for integer constants and floating-point constants are written separately and the commonality between them is extracted during the generation process. This increased modularity means that specification files are easier to write, read, and modify compared with a hand-written Java programs.

Thus, CSL is defined in a file named Translator.txt, which will be "compiled" by the JavaCC tool into a set of Java source class files of type .java. To generate a language, it is necessary to complete the following:

1. Options and class declarations
2. Specifying the lexical analyzer
3. Specifying the parser
4. Generating the parser and lexical analyzer

Having constructed the Translator.txt file, we invoke JavaCC on it. For example, under Microsoft Windows, using the “command prompt” program (CMD.EXE) we run JavaCC. This generates seven Java classes, each in its own file, produced by a Java compiler.

- *TokenMgrError* is a simple error class; it is used for errors detected by the lexical analyzer and is a subclass of *Throwable*.
- *ParseException* is another error class; it is used for errors detected by the parser and is a subclass of *Exception* and hence of *Throwable*.
- *Token* is a class representing tokens. Each *Token* object has an integer field *kind* that represents the kind of the token (BOTransaction, BOInput, etc) and a String field *image*, which represents the sequence of characters from the input file that the token represents.
- *SimpleCharStream* is an adapter class that delivers characters to the lexical analyzer.
- *TranslatorConstants* is an interface that defines a number of methods used in both the lexical analyzer and the parser.
- *TranslatorTokenManager* is the lexical analyzer.
- *Translator* is the parser.

By executing these classes in a Java Virtual Machine (JVM), we are able to translate from CSL to XML scripts.

7.4 Reverse Translation

Translating from XML scripts to CSL is necessary in order to make previously existing scripts available in CSL for maintenance. Translation in this direction is inherently simpler because XML is very easy to parse. We used Extensible Stylesheet Language Transformations (XSLT) to accomplish this. XSLT is a XML-based language that can be used to translate from XML to another text based language (here, the CSL). A processor is written in Java to perform the translation using the XSL file. Any changes to the XSL file do not need re-compiling of the processor, adding to the advantage of this approach.

8. Conclusions

SBML is intended for rapid, flexible prototyping of BML services. It has been developed into a well-rounded capability for generating Web services quickly and with a low error rate, based on a simple scripting language. While SBML is general enough to accept any XML-based input and work with any data model capable of representing the input, our implementations have focused on BML as the input language and JC3IEDM as the data model. SBML was used for this purpose in support of NATO MSG-2009 in 2008 and 2009.

SBML has been extended to support publish/subscribe and a condensed scripting language (CSL). Most recently it has been enhanced to support full XML namespaces for greater flexibility and multithreaded operation for improved performance.

A related development tool, the BML C2 GUI, was briefly introduced in this paper and will be described further in an SIW Fall 2010 paper.

References

[1] Carey, S., M. Kleiner, M. Hieb, and R. Brown, “Standardizing Battle Management Language – A Vital Move Towards the Army Transformation,” IEEE Fall Simulation Interoperability Workshop, Orlando, FL, 2001

[2] Pullen, J., D. Corner and S. Singapogu, “Scripted Battle Management Language Web Service Version 1.0: Operation and Mapping Description Language,” IEEE Spring 2009 Simulation Interoperability Workshop, San Diego CA, 2009

[3] Pullen, J., D. Corner and S. Singapogu, "Scripted Battle Management Language Web Service Version 2," IEEE Fall 2009 Simulation Interoperability Workshop, Orlando, FL, 2009

[4] Levine, S., L. Topor, T. Troccola, and J. Pullen, "A Practical Example of the Integration of Simulations, Battle Command, and Modern Technology," IEEE European Simulation Interoperability Workshop, Istanbul, Turkey, 2009

[5] Pullen, J. *et al.*, "An Expanded C2-Simulation Experimental Environment Based on BML," IEEE Spring Simulation Interoperability Workshop, Orlando, FL, 2010

[6] Sun Microsystems, "JAVA Message Service 1.1" April 12, 2002.

[7] World Wide Web Consortium, "XPath Language Version 1.0", November 16, 1999

[8] Corner, D., J. Pullen, S. Singapogu and B. Bulusu, "Adding Publish/Subscribe to the Scripted Battle Management Language Web Service," IEEE Spring 2010 Simulation Interoperability Workshop, Orlando FL, 2010

[9] US Department of Defense, "Interface Standard Common Warfighting Symbology MIL-STD-2525B w/Change 1", July 2005

Author Biographies

DR. J. MARK PULLEN is Professor of Computer Science at George Mason University, where he serves as Director of the C4I Center and also heads the Center's Networking and Simulation Laboratory. He has served as Principal Investigator of the XBML and JBML projects.

DOUGLAS CORNER is a member of the staff of the George Mason University C4I Center. He is the lead software developer on the SBML scripting engine.

SAMUEL SINGAPOGU is a member of the staff of the George Mason University C4I Center. He is the lead script developer on the SBML scripting engine.

BHARGAVA BULUSU is a member of the staff of the George Mason University C4I Center. He is a script developer on the SBML scripting engine. And primary developer of SBML translators between the XML script format and the Condensed Scripting Language.

MOHAMMAD ABABNEH is a Major in the Jordanian Air Force and a doctoral student in Information Technology at George Mason University. He is the primary developer of the BML C2 GUI.

Appendix 1: BNF Representation of Condensed Scripting Language

```
SKIP : { " " }
SKIP : { "\t"| "\r"| "\n"}

TOKEN :

{
    <assign: "Assign">
|   <BOTransaction: "BOTransaction">
|   <BOInput: "BOInput">
|   <abort: "Abort">
|   <call: "Call">
|   <put: "PUT">
|   <get: "GET">
|   <IfThen: "IfThen">
|   <IfThenElse: "IfThenElse">
|   <getList: "GETList">
|   <getRow: "GETRow">
|   <BOReturn: "BOReturn">
|   <BOReturnElement: "BOReturnElement">
|   <BOReturnElementTag: "BOReturnElementTag">
|   <HigherTagStart: "HigherTagStart">
|   <HigherTagEnd: "HigherTagEnd">
|    <Tag: “Tag”>
|   <relation: "EQ"|"NE"|"LT"|"GT"|"LE"|"GE"|"EQI"|">"|"=">
|   <comment: "//">
|   <Identifier: <lowercaseAndOtherCharacters>>
|   <variable: (<literalValue>|<Identifier>)+>
|   <literalValue: <Identifier>|””>
|   <lowercaseAndOtherCharacters: (["A"-"Z","a"-"z","0"-"9",".","_","/",":","<","-","*"])+>
|   < br: ";">
}
```

NOTE: Whitespace (any number of blanks and carriage returns) is the delimiter, unless other delimiter is indicated.

Appendix 2: XML Script Example

```
<?xml version="1.0" encoding="UTF-8"?>
<BusinessObjectInput xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="BusinessObjectTransactionInterpreterSchema.v0.18.xsd">
<!-- Fragment of code from WhatWhenPush -->
    <call>
        <boName>TaskeeWhoPush</boName>
        <anchorTag>TaskeeWho</anchorTag>
        <parameter>
            <workingVariable>task_act_id</workingVariable>
        </parameter>
    </call>
. . .
    <BusinessObjectTransaction>
        <transactionName>TaskeeWhoPush</transactionName>
        <parameter>task_act_id</parameter>
        <tableQuery>
            <databaseTable>unit</databaseTable>
            <queryAction>GET</queryAction>
            <resultName>unit_id</resultName>
            <columnReference>
                <columnName>formal_abbrd_name_txt</columnName>
                <businessObjectTag>UnitID</businessObjectTag>
            </columnReference>
        </tableQuery>
        <tableQuery>
            <databaseTable>act_res</databaseTable>
            <queryAction>PUT</queryAction>
            <columnReference>
                <columnName>act_id</columnName>
                <workingVariable>task_act_id</workingVariable>
            </columnReference>
            <columnReference>
                <columnName>act_res_index</columnName>
                <workingVariable increment="Yes">act_res_index</workingVariable>
            </columnReference>
            <columnReference>
                <columnName>cat_code</columnName>
                <literalValue>RI</literalValue>
            </columnReference>
            <columnReference>
                <columnName>authorising_org_id</columnName>
                <workingVariable>unit_id</workingVariable>
            </columnReference>
        </tableQuery>
        <tableQuery>
            <databaseTable>act_res_item</databaseTable>
            <queryAction>PUT</queryAction>
            <columnReference>
                <columnName>act_id</columnName>
                <workingVariable>task_act_id</workingVariable>
            </columnReference>
            <columnReference>
                <columnName>act_res_index</columnName>
                <workingVariable>act_res_index</workingVariable>
            </columnReference>
            <columnReference>
                <columnName>obj_item_id</columnName>
                <workingVariable>unit_id</workingVariable>
            </columnReference>
        </tableQuery>
        <BusinessObjectReturn >
            <BusinessObjectReturnElement>
                <tag>Result</tag>
                <literalValue>OK</literalValue>
            </BusinessObjectReturnElement>
        </BusinessObjectReturn>
    </BusinessObjectTransaction>
</BusinessObjectInput>
```