Appliponents: Extensions to UML Class and Component Diagrams

Vijai Mohan
vmohan@cse.uta.edu

Technical Report CSE-2004-15

This report was submitted as an M.S. CSE thesis.
The members of the Committee approve the master’s thesis of Vijai Mohan

Arthur Alexander Reyes  
Supervising Professor  

Lynn Peterson  

Carter Tiernan
APPLIPOMENTS: EXTENSIONS TO UML CLASS AND COMPONENT DIAGRAMS

by

VIJAI MOHAN

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

MAY 2004
ACKNOWLEDGEMENTS

I would like to acknowledge my supervising professor Dr. Arthur Reyes for his valuable guidance both technically and non-technically all through the research. I would also like to thank Dr. Lynn Peterson and Dr. Carter Tiernan for serving in my thesis committee and for their valuable suggestions. Suggestions and opinions from research colleagues, José Espino and Monica Nadkar were valuable. Last but not the least, I appreciate Dr. Puppala and other friends who have supported me during dull times and who have encouraged me throughout graduate school. I would like to thank the Department of Computer Science and Engineering, Department of Civil Engineering and the College of Engineering for the financial assistance provided through out my degree.

April 02, 2004
ABSTRACT

APPLIPONENTS: EXTENSIONS TO UML CLASS AND COMPONENT DIAGRAMS

Publication No. ______

Vijai Mohan, M.S.
The University of Texas at Arlington, 2004

Supervising Professor: Arthur Alexander Reyes

We propose “appliponents”—extensions to UML Class and Component Diagrams that will enable them to better accommodate extant, non-object-oriented, and legacy software. Appliponents are applicative, dataflow-oriented representations of subprograms (e.g., functions, procedures, methods, operations, operators, etc.) and of standalone executable programs (i.e., appliponents model executable software artifacts that features a single entry point). Appliponent decomposition diagrams (ADDs) are similar to notations used by Simulink, LabVIEW, and SoftWIRE, but appliponents feature a powerful mechanism for making what were traditionally implicit interfaces (global data and I/O) fully explicit through an unique orthogonal classification scheme. ADDs lend more rigorous software design discipline for projects relegated to the category of “glue coding”. ADDs are therefore applicable to enterprise application integration (EAI) projects where the customer is unwilling to pay (1) for a complete, unified data model; (2) to adopt a standard data model; (3) for re-engineering extant software applications; or (4) for a common technical infrastructure.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................................................................................ iv

ABSTRACT ................................................................................................................................ v

Chapter

1. INTRODUCTION ....................................................................................................... 01

2. LITERATURE REVIEW .......................................................................................... 10

3. APPLIPONENT DIAGRAMS .................................................................................. 20

4. APPLIPONENT DECOMPOSITION DIAGRAM (ADD) EXAMPLE .......................... 34

5. CONCLUSIONS AND FUTURE WORK .................................................................. 39

REFERENCES .................................................................................................................... 44

BIOGRAPHICAL INFORMATION ....................................................................................... 48
LIST OF FIGURES

Figure                                      Page
2.1  MSW Logo – A Modern Version of LOGO ................................................... 11
2.1  Screen Shot of Karel Programming Environment ........................................... 12
2.2  Sample Lego Program ..................................................................................... 12
2.3  FDDL Example ............................................................................................... 15
2.4  SoftWIRE ........................................................................................................ 18
3.1  Example Class Notation with Appliponent Nodes ............................................. 31
3.2  Package and Component Diagrams Showing Appliponents ............................. 32
3.3  Appliponent Diagram- Symbols with Key ....................................................... 33
4.1  Genetic Algorithm Work Flow ....................................................................... 35
4.2  Proposed Workflow in MArSHLAnd – Original Representation ................. 36
4.3  Appliponent Diagram for MArSHLAnd- New Representation ....................... 38
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Qualitative Summary of Related Work</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Activation Input</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Interaction Input</td>
<td>27</td>
</tr>
<tr>
<td>3.3 Interaction Output</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Passivation Output</td>
<td>29</td>
</tr>
<tr>
<td>5.1 Table 5.1 Summary of Rules Satisfied By ADD</td>
<td>40</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1. Introduction

We propose and evaluate a novel model for “black box” enterprise application integration (EAI). EAI is defined as “…the cooperation of disparate software components to implement business rules in a distributed environment.” [1] EAI is important because consolidation of enterprises mandates that their Information Technology (IT) be consolidated as well. If company A and company B merge (or company A purchases company B), then the IT of company A needs to be merged (or integrated) with the IT of company B. Often a enterprise’s IT represents a significant fraction of the enterprise’s value to its stockholders [2]. Such IT is too valuable to ignore. As enterprises grow in size, the risk of data duplication and application feature duplication increases. This has led to the need to share data between enterprise applications in a reliable way. Although manual intervention to share data between applications is possible, the volume of data to be handled shows the need for an automated solution. Furthermore, collaboration efforts to set standards like EDI (Electronic Data Interchange—the old standard for business-to-business electronic data interchange over a network [35]) have not been successful enough to completely solve the problem as the application domains and expectations keep increasing continuously.
An EAI project requirements document will specify (1) the suite of extant software applications to be integrated and (2) the new or modified features to be present in the integrated suite of applications. “Black box” means that we do not want to make intrusive changes to the extant software applications to be integrated—we want to integrate them as-is. Note that it may be necessary to analyze the insides of extant software applications in order to better understand their implicit interfaces (more about this is discussed later).

EAI is difficult because:

- WRAPPING: Wrapping is the process of encapsulating software with new interfaces that are easier to (re)use than the original interfaces. An interface is the façade for any means of invoking application features, observing the state of an application, or controlling the state of an application. The wrapping activity attempts to make as few changes to the software as possible (i.e., it endeavors to treat applications as black boxes). It is desirable to wrap a software application when it is apparent that any significant internal change (i.e., re-engineering) to the application will significantly break its functionality. Wrapping is difficult because you may have to reverse engineer proprietary or badly-designed data structures or file formats in order to make the features of those data available for other software applications in the EAI suite.

- RE-ENGINEERING: Re-engineering is the sequential process of reverse engineering, restructuring, and forward engineering. Re-engineering aims to provide software applications with easier-to-(re)use interfaces than they possessed before, greater fault-tolerance, greater maintainability, testability, usability, etc. Re-engineering
is difficult because there may exist numerous, hidden data and control dependencies within and between the software applications being re-engineered. Such hidden dependencies may not be discovered until late in the development process when integration and system testing reveals that certain sequences of feature invocations cause the application to terminate abnormally.

- **SCOPE OF DATA UNIFORMITY:** The customer of the EAI project may not be willing to fund development of a unified data model that covers all the features of all the applications to be integrated. Instead the customer of the EAI project will have a prioritized list of requirements. The highest-priority requirements will use only a small subset of data available in the applications to be integrated. A small, but unified data model must be defined for these requirements. As the number of requirements to be implemented increases, the unified data model becomes larger. As funds become available, implementing requirements in strict succession will not be cost-optimal: The previous unified data model will have to be restructured and the changes trickled down to all the software applications that use the data. Optimally, you want to generate a unified data model in the first iterations of the EAI project. If appropriate, you want to use standardized data models that cover a broad part of the enterprise. RosettaNet [4] is an example of a standardized data model for electronic commerce.

- **TECHNOLOGY CHOICES:** Choice of available technology is enormous; hence deciding on the right approach may be challenging task.
• **INACCURATE EXPECTATIONS:** An EAI project requirements document might inaccurately assume that a software application possesses features or can support performance loads that it really does not possess or cannot support.

• **TIMELINESS:** If you take too long to perform your EAI project, your chosen EAI technology may become obsolete before you are finished.

• **SOLUTION-INDEPENDENT PROBLEM MODELING:** There is no standard way of representing an EAI problem without implicitly stating the form of the solution. Currently, the way you specify the problem depends on the chosen solution technology. For example, if you attempt to model an EAI problem using Unified Modeling Language (UML) [26], you will have trouble representing extant, non-object-oriented software applications.

1.1.1 **Introducing Appliponents**

In response to the difficulties described in section 1.1, we propose to extend UML 1.8 with a new modeling element known as an “appliponent”. An appliponent (i.e., “application-as-component”—a way of considering entire software applications as components in new, larger software applications) is a graphical representation for subprograms and standalone executable files (i.e., “*.exe”) with purely combinational/functional semantics. Recall any sequential machine can be considered as a combinational function if you think about its input as being a sequence of data and its output as being a sequence of data. Appliponents can do this because they make all implicit interfaces (e.g., side effects and I/O) fully explicit.
Appliponent nodes that represent subprograms can appear in UML 1.8 class diagrams as follows: Appliponents nodes representing methods of a class appear in the lower part of the class’s rectangle instead of the usual textual declaration of the method. Appliponents representing global procedures and functions appear as nodes in class diagram packages or “utility classes”—classes that do not have instance objects but are simply containers for subprograms.

Appliponents that represent standalone executable program files can appear in UML 1.8 component diagrams. In all the previously described extensions to UML 1.8 diagrams, each node represents the *declaration* of an appliponent. In all three of the above notations, there are no interconnections shown between appliponent nodes.

We also propose an entirely new kind of diagram to supplement UML 1.8 component diagrams: appliponent decomposition diagrams (ADDs). An ADD shows the internal workings of an appliponent as a directed graph of appliponent nodes connected by dataflows. In ADDs, each node represents an *invocation* of the named appliponent (thus there can be more than one node with the same name). ADDs were inspired by notations such as Functional Design Decomposition Language (FDDL) [11], Infinity [28], SIMULINK [27], LabVIEW [29], VisSim (formerly known as MathConnex) [36], and more recently by SoftWIRE [12]. However ADDs possess a superior facility for representing global data, I/O, and other implicit interfaces than any of these other notations.

We believe the appliponent representation is applicable to EAI projects where any of the following conditions hold.
• The customer is unwilling to pay for a complete, unified data model.

• The customer is unwilling to pay to adopt a standard data model, e.g., RosettaNet [4].

• The customer is unwilling to pay for re-engineering extant software applications, but is willing to pay for some analysis of extant software applications.

• The customer is unwilling to pay for a common technical infrastructure (e.g., a distributed object technology such as Enterprise Java Beans (EJB), Common Object Request Broker Architecture (CORBA), or Microsoft .NET).

Thus the appliponent representation is a graphical design notation for EAI projects that are often relegated to the category of “glue coding”. We draw appliponents as rectangles in our extensions to UML 1.8. The left side of the appliponent is for data that must be provided when the appliponent starts execution. The right side of the appliponent is for data that are made available when the appliponent ends execution. The top side of the appliponent is for global data inputs. The bottom side of the appliponent is for side-effects.

Each side of an appliponent node is further divided into four different kinds of interfaces—(controlled, memory), (controlled, stream), (uncontrolled, memory), and (uncontrolled, stream). The memory/stream dichotomy is dictated by location where data resides like memory space of an OS process or in a file or stream. The
controlled/uncontrolled dichotomy is dictated by whether a written grammar, interface, or schema exists that can validate the syntax/semantics of the data.

Appliponents are most reminiscent of “components” used in the SoftWIRE™ integrated development environment (IDE) [3]. As in SoftWIRE™, you design a new application by “wiring” the output ports on the right or bottom sides of a component with the input ports on the top or left sides of a component. With ADDs, control flow dependencies are automatically derived from dataflow dependencies between appliponents whenever possible. Thus control flow often does not have to be explicitly represented in the appliponent representation. In future work we will develop a code generator that takes an ADD as input and generates complete, correct source code as output.

1.2 Research Objectives

The main research objectives of this thesis are as follows.

- With some level of precision, define the following terms: appliponent, controlled data, uncontrolled data, data in memory, data in streams, input data, interaction input data, interaction output (i.e., side effect) data, and passivation data. Give adequate, clear examples of each kind of data.

- Give examples of appliponent-based extensions to UML 1.8 class diagrams and component diagrams.

- Give examples of appliponent decomposition diagrams (ADDs): a graphical notation for “wiring” appliponents together into a “black box” EAI solution.
• Qualitatively evaluate our work with respect to the corpus of related work in dataflow-oriented visual modeling/design/programming notations and tools.

1.3 Thesis Organization

This thesis is composed of five chapters: introduction (chapter 1), literature review (chapter 2), analysis and development of framework (chapter 3), case studies (chapter 4), summary, conclusions and future work (chapter 5).

Chapter 1 provides the introduction, objective and thesis organization. The problem definition and objectives of the research are included in this chapter.

Chapter 2 provides a summary of various dataflow-oriented visual programming tools currently used in the software industry. Rules to be satisfied by any visual representation to be successful is reviewed in this section [5]. Finally, existing visual programming application/representation like FDDL and SoftWIRE are presented.

Chapter 3 presents the classification of data based on several critical parameters, our definitions for appliponents and appliponent diagrams. Rules for classifying each data input in to the appliponent, and output of an appliponent is detailed with examples. It is this unique, orthogonal classification of data that enables appliponents to make what used to be considered implicit interfaces fully explicit.

Chapter 4 details a case study implemented using appliponents. Detailed summary of the scenario is presented with (Unified Modeling Language) UML diagrams and improved ADD diagrams.
Chapter 5 provides summary and major conclusions of the research study and results. Future research needs and directions are mentioned.
CHAPTER 2
LITERATURE REVIEW

2.1 Why Visual Programming

Visual programming represents the idea of developing software programs by sequencing meaningful symbols in a logical sequence. The idea of visual programming when introduced in 1950 didn’t have much success due to the lack of tools to develop powerful graphical interfaces [6]. The first powerful attempt was the programming language LOGO (shown in figure 2.1) from the Massachusetts Institute of Technology, which used “turtle” as the user interface and LISP as the transformation language. Even though the language created a lot of enthusiasm, it was not used widely in industry or academia. Currently visual programming is used widely in many areas and most of them use a data flow notation. Some examples are

- ROO, Turtle graphics and various versions of Karel (shown in figure 2.2) are currently used in various universities to teach programming concepts to students. [7] [8] [9].

- Lego MindStorms provides a visual programming interface to program the Lego kits (shown in figure 2.3).

- Visual programming technique is used extensively in simulation packages like Simulink and data acquisition systems like LabVIEW use visual programming extensively.
Figure 2.1 MSW Logo – A Modern Version of LOGO [37]

Currently most visual programming notations and tools confine solutions to a specific domain where the dataflow paradigm is a good fit. Furthermore, there are not many tools or representations for integration or modeling disparate software applications. SoftWIRE can satisfy some of these requirements and is discussed later. We also investigated BizTalk™ [32] as a potential tool for this purpose. Within a week’s inspection, it was obvious that the amount of effort involved in developing the solution using BizTalk would be significant. BizTalk’s constraints were very limiting and in some cases, BizTalk didn’t have the capability to correctly represent our deeply-nested data.
Figure 2.2 Screen Shot of Karel Programming Environment [30]

Figure 2.3 Sample Lego Program [31]
2.2 Characteristics of Visual Representation

In order for software visualization systems to support large-scale software development and maintenance, visualization systems are analyzed from five dimensions [10]. The five dimensions are

Tasks -Why is Visualization needed?

Audience-Who will use the visualization?

Target-What is the data source to be represented?

Representation- How to represent it?

Medium- Where to represent it?

Further, Shneiderman [5] presented seven high-level requirements for the representation-aspect of visualization that every representation tool must support. These needs are as follows.

- Overview: The representation must be able to provide an overview of the entire collection of data.
- Zoom: The representation should have the ability to focus on a part of the collection without losing the state of the entire collection.
- Filter: The representation should have an ability to filter items of lesser interest.
- Details-On-Demand: The representation should have provision to see more detailed information regarding any component.
- Relate: The representation should provide ways to show the relationship between various components.
• History: Store the actions of the users.

• Extract: Allow extracting subcomponents in the representation and a facility to work on them individually.

2.3 FDDL

FDDL is a graphical language supporting design of pure functions and functional decomposition [11]. Any program is represented in FDDL as an abstract syntax tree. Every node in the tree represents either an operator or a function call. In FDDL, all the nodes exhibit purely functional behavior, i.e. the nodes are stateless. Every node can contain multiple input ports and one output port. Every edge in FDDL connects a data output to data inputs. All operators like + or * are completely associative in nature and operate on any nonzero number of inputs unlike the traditional operators which operate on two operands only. FDDL language is shown in figure 2.4
2.4 SoftWIRE [12]

SoftWIRE was introduced in 2001 as a plug-in for Microsoft’s Visual Studio to generate source code from graphical representations. SoftWIRE is probably the first general-purpose graphical programming tool with significant capabilities and potential.

SoftWIRE supports the “communicating processes” architectural style. Each component is an ActiveX object [13]. Connectors represent defined communication channels for events and data. Any ActiveX object (and there are thousands) can appear in an application’s design. Any required programming is done in Visual Basic. SoftWIRE is shown in figure 2.5

Figure 2.4 FDDL Example of the Controller for a Genetic Algorithm [17]
Table 2.1 Qualitative Summary of Related Work

<table>
<thead>
<tr>
<th>Feature</th>
<th>ADD</th>
<th>FDDL</th>
<th>SoftWIRE™</th>
<th>Infinity</th>
<th>Simulink</th>
<th>LabVIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation</td>
<td>Purely Functional</td>
<td>Purely Functional</td>
<td>Nodes can be either functional (i.e., “+”) or have internal states.</td>
<td>functional or stateful.</td>
<td>functional or stateful.</td>
<td>functional or stateful.</td>
</tr>
<tr>
<td>Evaluations</td>
<td>Truly parallel</td>
<td>Evaluations are truly parallel</td>
<td>Sequential</td>
<td>Sequential</td>
<td>Sequential</td>
<td>Sequential</td>
</tr>
<tr>
<td>Categories of data addressed</td>
<td>16 categories (see text)</td>
<td>Input and Output only</td>
<td>Input, Output, Node Activation, Exceptions, Activate other nodes</td>
<td>Input, Output, Control</td>
<td>Input, Output, Initial Condition</td>
<td>Input, Output and Global Data</td>
</tr>
<tr>
<td>Are Nodes Decomposable</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Dataflow-oriented?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Control flow-oriented?</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Nodes</td>
<td>Appliponents</td>
<td>Operators or Functions</td>
<td>ActiveX component or functions or operators</td>
<td>Code Blocks or Logical</td>
<td>Execution Blocks</td>
<td>Operators, Functions and Virtual Instruments</td>
</tr>
<tr>
<td>Data types</td>
<td>simple and complex data types (SOAP)</td>
<td>simple data types and tuples</td>
<td>Simple, Complex and Custom data types</td>
<td>Simple data types and Arrays</td>
<td>Simple data types &amp; matrices</td>
<td>Custom and Simple Data types</td>
</tr>
</tbody>
</table>
With tools such as this, application development proceeds as follows.

1. Import the component libraries onto the editor’s palette.
2. Drag the specific components you want onto the editor’s canvas.
3. The interface ports of each component become visible. “Tool tips” describe the port’s name, type, options, etc.
4. Rename components as needed.
5. Connect some of the output ports of some components on the canvas to the input ports of some components also on the canvas.
6. Automatically check that connections are valid.
7. As appropriate, ensure that the new application has an appropriate interface that will allow it to become a reusable component in later, unforeseen applications.
8. Compile, link, and run the application.
These tools use a powerful metaphor for application composition from preexisting components: “Wire the components together”. They provide excellent support for component composition, and design decomposition to a lesser degree. For some reason, tools such as these are very popular in engineering and science fields outside of computer science and engineering, but virtually unknown within computer science and engineering. The answer for the disparity could be a tendency by computer scientists/engineers to enjoy programming more, but this could be a research question in its own right.

Tools such as Simulink, LabVIEW, VisSim, SoftWIRE, and enable high productivity because the components they manipulate satisfy a lengthy list of requirements, defined in what are called “software component architectures”. CORBA [14] and COM are examples of standardized, general-purpose, software component architectures. Component architectures such as these support “binary interoperability”, “reflection”, “distribution”, and other interesting features. Binary interoperability allows any two components to interoperate, regardless of which programming languages were used in their source code. Reflection allows a component to be aware of facts about itself, such as its declared interface, and to tell other components those facts.

So when will all software development be performed under the metaphor of an appropriate architectural style and with easily-reusable components? Under current approaches, this will probably never occur. The tools described earlier and the component architectures that support them are most appropriate for developing new applications from existing components. The tools and component architectures
described earlier offer little hope for older software applications. While it is possible to transform an old-but-useful application into a collection of reusable COM components, doing so with current methods is not likely to be cost-effective. Thousands of old-but-useful software applications were developed as vertically-integrated “stovepipes”: They do a few things, but do them very well. They were not designed to offer reusable, robust, generic services to unforeseeable kinds of clients.
CHAPTER 3
APPLIPONENT DIAGRAMS

3.1 Introduction

The focus of this research is to provide a visual notation that can clearly show the details of implementations in software architectures or designs. Dataflow diagrams tend to be too complex to track for big projects. There is very limited provision to show state of objects, classes and interaction between those classes precisely using the data flow notation.

We present Appliponent diagrams in this thesis, as a potential extension to UML notations to explicitly model implicit interfaces.

A brief description of the symbolic notation along with the ideas to create a CASE tool is detailed in Section 3.2

3.2 Appliponent Diagram Definitions

Appliponent decomposition diagram is a design notation to show data flow from between appliponents. An ADD shows the internal logical structure of an appliponent as a directed graph of appliponent nodes connected by edges representing dataflow. To clearly explain an appliponent diagram we define some terms:

Appliponent (“application-as-component”): An appliponent is a fully modular, combinational/applicative/purely functional *representation* for a defined structure of executable program statements. An appliponent can be any executable file or
subprogram exported by a programming library file. An appliponent represents a software feature(s) which is available for binary-compatible reuse in new, larger software applications.

For appliponents that represent subprograms (i.e., procedures and functions), “appliponent execution” begins with the moment the activation record for the appliponent is placed on the memory stack and ends at the moment the activation record is popped off the memory stack. For appliponents that represent standalone executable program files, “appliponent execution” begins with the moment the OS creates the process for the appliponent and ends at the moment when the OS terminates the process for the appliponent.

An executable file is a file that the OS interprets by creating a process and setting the instruction pointer to an initial location in the code segment of that process. Examples of executable files include “commands”, “Load modules”, “standalone executable files” (*.exe), “batch command files” (*.com), etc.

Examples of programming library files include dynamic link library (*.dll) files, compiled but unlinked object (*.obj) files, and executable files that additionally export programming libraries (e.g., the Excel programming library (exported as COM interfaces) by C:\Program Files\Microsoft Office\Office10\EXCEL.EXE). Microsoft Excel exemplifies a suite of subprogram appliponents nested within an executable program appliponent.

To ensure pure functional behavior, appliponents explicitly represent all relevant data used or changed by each execution of the appliponent—formal
parameters, returned values, command-line arguments, command-line switches, optional arguments, private attributes, global variables and constants, environment variables, stdin, stdout, redirected or piped input or output, keyboard streams, mouse streams, video streams, device I/O streams, network streams, etc.—even if those interfaces are not directly accessible by the program that calls the appliponent. Appliponents model their interfaces as formal parameters. Each parameter falls into at least one of 16 categories, which we will describe later. The interface of the appliponent is discovered by reading documentation describing the appliponent, analyzing the dataflow anomalies in the appliponent’s source code, and any other source of relevant information. The relevance of data is judged by the programmer who wants to use the appliponent. Now we describe formally all the interfaces to an appliponent.

Left Side: Activation/Input data: Represents all data that are referenced before execution of the first statement in the appliponent.

Right Side: Passivation/Output data: Represents explicit data that are created or made available as a result of execution of the appliponent. Passivation for a program can be assumed to be the passive state when the amount of new resources like CPU cycles, memory etc is minimal in its life cycle. Passivation can be assumed to be the state of the program before termination.

Top Side: Interaction/Input data: Represents explicit and implicit data read by the appliponent during execution.
Bottom Side: Interaction/Output data: Represents explicit and implicit data created or changed by the appliponent during execution (i.e., side effects). Notice that interaction/output data can also be interaction/input data and vice versa.

Each side of the appliponent’s rectangle supports four categories of data. Based on the ease with which the semantics of appliponent’s input and output data can be determined, we classify such data as Controlled and Uncontrolled.

Controlled: The semantics of Controlled actual data values can be unambiguously validated with minimal human effort. Validation can be accomplished by invoking an extant form input validator or parser, referencing a data type interface specification that features sufficient, exported operations to meaningfully interpret instances of the data type, etc. The type specification of Controlled data must either reference the specification of the grammar using Uniform Resource Indicator (URI) [25] or else list the grammar itself.

Uncontrolled: The syntax and semantics of Uncontrolled actual data values can be unambiguously determined only with significant human effort. Such effort may require the definition of a grammar and generation of a parser, wrapping naked, proprietary C data structures within well-encapsulated programming interfaces, discovering the regular expression characterizing the valid set of values a string-typed datum can take on, etc.

Data of type “String” are usually categorized as Uncontrolled, even though a well-encapsulated String data type exists. The extant String data type does not export operations that decide whether actual String values are valid for the appliponent in
question. If the data can be expressed with little effort as a Simple Object Access Protocol (SOAP) [15] data type, then it would ideally be classified somewhere between Controlled and Uncontrolled. Future Work will refine the nominal Controlled/Uncontrolled measurement scale into more a realistic interval measurement scale or ratio measurement scale.

Based on the location where data resides, it must be categorized as “data in stream” or “data in memory”.

Data in Memory: Data residing in the private storage space allocated by the operating system for a specific process or for many processes.

Data in Streams: A stream is a sequence of characters. A stream does not need to be allocated to a process. All data not categorized as residing in memory reside in streams. Every platform Software Development Kit (SDK) (e.g., JDK, .NET platform SDK, Windows platform SDK) has its own type or class hierarchy for streams (and files). Such hierarchies are often rooted with a class named “Stream”. Given the appliponent under examination, we consider the platform SDK(s) that was/were used to create the appliponent, if this information is available (otherwise we consider the most appropriate SDK(s) available to us). Streamed data are those that are reasonably instances of the most appropriate platform SDK’s “Stream” hierarchy.

Combining the above discussed orthogonal classifications we can classify data in four quadrants as Memory/Controlled, Memory/Uncontrolled, Stream/Controlled and Stream/Uncontrolled. Commonly used data is classified based on these parameters in table 3.1.
Read the table with the belief that all the data examples in the “Controlled” rows possess immediately validatable semantics. Read the table with the belief that all the data examples in the “Uncontrolled” rows do not possess immediately validatable semantics. We say this because the examples often use the exact same words. These details are omitted from the table to preserve space.

Any examples of data below listed for subprograms are also examples for executable programs. The converse is not necessarily true.
### Table 3.1 Activation Input

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory/Controlled</td>
<td>- If the appliponent is a subprogram,</td>
</tr>
<tr>
<td></td>
<td>o Formal “in” and “in out” parameters residing in memory of a process (This can be a Stream handle.)</td>
</tr>
<tr>
<td></td>
<td>▪ Sort(intList : in List(Int))</td>
</tr>
<tr>
<td></td>
<td>- If the appliponent is an executable program file,</td>
</tr>
<tr>
<td></td>
<td>o Command line Arguments like basic data types</td>
</tr>
<tr>
<td></td>
<td>▪ multiply.exe 1 3 5.0</td>
</tr>
<tr>
<td></td>
<td>▪ time.exe –timezone CST</td>
</tr>
<tr>
<td>Stream/Controlled</td>
<td>- If the appliponent is a subprogram,</td>
</tr>
<tr>
<td></td>
<td>o Formal “in” and “in out” parameters which point to the contents of a Stream</td>
</tr>
<tr>
<td></td>
<td>- If the appliponent is an executable program file,</td>
</tr>
<tr>
<td></td>
<td>o Command line parameters to an URI pointing to a file</td>
</tr>
<tr>
<td></td>
<td>▪ <a href="http://fddl2/build.xml">http://fddl2/build.xml</a></td>
</tr>
<tr>
<td>Stream/Uncontrolled</td>
<td>- If the appliponent is a subprogram,</td>
</tr>
<tr>
<td></td>
<td>o Formal “in” and “in out” parameters which point to the contents of a Stream</td>
</tr>
<tr>
<td></td>
<td>- If the appliponent is an executable program file,</td>
</tr>
<tr>
<td></td>
<td>o Command line parameters to an URI pointing to a stream</td>
</tr>
<tr>
<td></td>
<td>▪ <a href="http://fddl2/val.dat">http://fddl2/val.dat</a></td>
</tr>
<tr>
<td>Memory/Uncontrolled</td>
<td>- If the appliponent is a subprogram,</td>
</tr>
<tr>
<td></td>
<td>o Formal “in” and “in out” parameters residing in memory of a process (This can be a Stream handle.)</td>
</tr>
<tr>
<td></td>
<td>▪ func(s : in String)</td>
</tr>
<tr>
<td></td>
<td>- If the appliponent is an executable program file,</td>
</tr>
<tr>
<td></td>
<td>o Command line parameters</td>
</tr>
<tr>
<td></td>
<td>▪ test.exe –n abc.123 def.456</td>
</tr>
</tbody>
</table>
### Table 3.2 Interaction Input

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory/Controlled</strong></td>
<td>• If the appliponent is a subprogram,</td>
</tr>
<tr>
<td></td>
<td>o All class attributes, global variables, or static variables such</td>
</tr>
<tr>
<td></td>
<td>that after activation, for any path through the appliponent’s CFG, the</td>
</tr>
<tr>
<td></td>
<td>first “action” on the data is a reference.</td>
</tr>
<tr>
<td></td>
<td>o Referencing a stream port handle</td>
</tr>
<tr>
<td></td>
<td>• If the appliponent is an executable program file,</td>
</tr>
<tr>
<td></td>
<td>o Use of operating system properties or any shared variable between</td>
</tr>
<tr>
<td></td>
<td>processes come in this category.</td>
</tr>
<tr>
<td></td>
<td>- System properties like System.properties.getProperty(&quot;path&quot;)</td>
</tr>
<tr>
<td><strong>Stream/Controlled</strong></td>
<td>• If the appliponent is a subprogram,</td>
</tr>
<tr>
<td></td>
<td>o The value obtained by calling methods in a web service (By definition</td>
</tr>
<tr>
<td></td>
<td>such data are valid against an XML schema.)</td>
</tr>
<tr>
<td></td>
<td>o Referencing the data in a stream port</td>
</tr>
<tr>
<td></td>
<td>• If the appliponent is an executable program file,</td>
</tr>
<tr>
<td></td>
<td>o If it uses any type of well-defined network protocol like HTTP or FTP,</td>
</tr>
<tr>
<td></td>
<td>the interactions are well defined and can be easily validated against</td>
</tr>
<tr>
<td></td>
<td>the RFC’s protocol specifications</td>
</tr>
<tr>
<td><strong>Stream/Uncontrolled</strong></td>
<td>• If the appliponent is a subprogram,</td>
</tr>
<tr>
<td></td>
<td>o Any uncontrolled data read from configuration file, temp files or other</td>
</tr>
<tr>
<td></td>
<td>log files</td>
</tr>
<tr>
<td></td>
<td>• If the appliponent is an executable program file,</td>
</tr>
<tr>
<td></td>
<td>o If it uses any type of propriety network protocol like Real™, the</td>
</tr>
<tr>
<td></td>
<td>interactions are not well-defined and cannot be easily validated.</td>
</tr>
<tr>
<td></td>
<td>o Keystrokes</td>
</tr>
<tr>
<td></td>
<td>o Mouse data stream</td>
</tr>
<tr>
<td></td>
<td>o Scripting actions for GUI Windows script or Linux script to generate</td>
</tr>
<tr>
<td></td>
<td>keystroke and mouse event streams</td>
</tr>
<tr>
<td></td>
<td>o In applications featuring a console interface,</td>
</tr>
<tr>
<td></td>
<td>- “redirecting” the input keystrokes from a text file</td>
</tr>
<tr>
<td></td>
<td>- “piping” the output of another program into the current executable</td>
</tr>
<tr>
<td></td>
<td>program</td>
</tr>
<tr>
<td><strong>Memory/Uncontrolled</strong></td>
<td>• If the appliponent is a subprogram,</td>
</tr>
<tr>
<td></td>
<td>o Any class variable or global variable or static variable used by a</td>
</tr>
<tr>
<td></td>
<td>subprogram</td>
</tr>
<tr>
<td></td>
<td>• If the appliponent is an executable program file,</td>
</tr>
<tr>
<td></td>
<td>o Shared variables like system time, screen resolution etc</td>
</tr>
<tr>
<td>Table 3.3 Interaction Output</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Memory/Controlled</strong></td>
<td></td>
</tr>
<tr>
<td>• If the appliponent is a subprogram,</td>
<td></td>
</tr>
<tr>
<td>o Changes to class variable or global variable or static variable made by a subprogram</td>
<td></td>
</tr>
<tr>
<td>o Writing data to a stream port handle</td>
<td></td>
</tr>
<tr>
<td>• If the appliponent is an executable program file,</td>
<td></td>
</tr>
<tr>
<td>o Modification system properties or any shared variable between processes come in this category.</td>
<td></td>
</tr>
<tr>
<td>▪ System.properties.setProperty(“path”,”c:\newpath”)</td>
<td></td>
</tr>
<tr>
<td><strong>Stream/Controlled</strong></td>
<td></td>
</tr>
<tr>
<td>• If the appliponent is a subprogram,</td>
<td></td>
</tr>
<tr>
<td>o Shell based authentication system where it sends the user name and passwords to the remote host</td>
<td></td>
</tr>
<tr>
<td>o Writing data to a stream port</td>
<td></td>
</tr>
<tr>
<td>If the appliponent is an executable program file,</td>
<td></td>
</tr>
<tr>
<td>o Any web server responds with well-defined HTTP “response” to the request</td>
<td></td>
</tr>
<tr>
<td><strong>Stream/Uncontrolled</strong></td>
<td></td>
</tr>
<tr>
<td>• If the appliponent is a subprogram,</td>
<td></td>
</tr>
<tr>
<td>o Response to proprietary network protocols</td>
<td></td>
</tr>
<tr>
<td>If the appliponent is an executable program file,</td>
<td></td>
</tr>
<tr>
<td>o Any GUI Interaction event causing screen to update continuously (We concern ourselves with outputs to video because “screen scrapers” can be used to “scrape” data from video when you have a executable program that does not programmatically export the desired data.)</td>
<td></td>
</tr>
<tr>
<td>o In applications featuring a console interface,</td>
<td></td>
</tr>
<tr>
<td>▪ “redirecting” the output text to a text file</td>
<td></td>
</tr>
<tr>
<td>▪ “piping” the output of the current executable program to another program</td>
<td></td>
</tr>
<tr>
<td><strong>Memory/Uncontrolled</strong></td>
<td></td>
</tr>
<tr>
<td>• If the appliponent is a subprogram,</td>
<td></td>
</tr>
<tr>
<td>o Changes to class variable or global variable or static variable made by a subprogram</td>
<td></td>
</tr>
<tr>
<td>• If the appliponent is an executable program file,</td>
<td></td>
</tr>
<tr>
<td>Modification to system properties or any shared variable between processes come in this category.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 3.4 Passivation Out

<table>
<thead>
<tr>
<th>Category</th>
<th>Conditions</th>
</tr>
</thead>
</table>
| Memory/Controlled | - If the appliponent is a subprogram,  
|                   |   o Formal “out” and “in out” parameters residing in memory of a process (This can be a Stream handle.)  
|                   |     ▪ GetTime(localTime : out Time)  
|                   |   o Function Return values  
|                   |     ▪ GetTime() : Time  
|                   | - If the appliponent is an executable program file,  
|                   |   o Clearly defined C return codes  
|                   |   o Any controlled variable (like mutex) shared between process remains in the memory until a memory reference to that variable exists. Changes by a process to these variables persist in memory even after termination. |
| Stream/Controlled | - If the appliponent is a subprogram,  
|                   |   o Formal “out” and “in out” parameters which point to the contents of a Stream  
|                   |   o The value returned by any method in a web service (By definition such data are valid against an XML schema.)  
|                   | - If the appliponent is an executable program file,  
|                   |   o A stream generated by any kind of code generator |
| Stream/Uncontrolled | - If the appliponent is a subprogram,  
|                   |   o Formal “out” and “in out” parameters which point to the contents of a Stream  
|                   |     ▪ println(“I am writing to stream %d”, i);  
|                   | - If the appliponent is an executable program file,  
|                   |   o DAT file, CSV files, Streams in proprietary format |
| Memory/Uncontrolled | - If the appliponent is a subprogram,  
|                   |   o Formal “out” and “in out” parameters residing in memory of a process (This can be a Stream handle.)  
|                   |     ▪ Func(s : out String)  
|                   |     ▪ toString() : String  
|                   | - If the appliponent is an executable program file,  
|                   |   o C return codes when the meaning of each code is NOT clearly defined  
|                   |   o Any uncontrolled variable (like mutex) shared between process remains in the memory until a memory reference to that variable exists. Changes by a process to these variables persist in memory even after termination. |
3.3 Diagram Notations and Conventions

Appliponents are modeled in two kinds of diagrams:

- UML Class Diagrams and Component Diagrams
- Appliponent Decomposition Diagrams (ADDs).

The static structure of the application is shown using the UML class diagrams, but textual method declaration is replaced with appliponent (subprogram) declarations. UML component diagrams embed appliponents in them like any other component. The extended static UML diagrams are presented in figures 3.1, 3.2. All the implicit interactions between the class variables and methods are explicitly shown as interaction inputs and outputs.

In an ADD, each node represents a call to the named appliponent. Each call is applicative (i.e., purely functional) and hence state variables enter and leave the system during the execution of the appliponent. All the arrows represent the data flow and show the sequential flow of data in and out of an appliponent. A list of recommended symbols along with its key is presented in figure 3.3
Private energy: int
Protected lArm: Arm
Protected rArm: Arm
Protected position: Position
Private mySensors: Sensor[]

Figure 3.1 Example Class Notation with Appliponent Nodes
Figure 3.2 Package and Component Diagrams Showing Applitonets
Figure 3.3 Appliance Diagram - Symbols with Key
CHAPTER 4

APPLIPONENT DECOMPOSITION DIAGRAM EXAMPLE (ADD)

4.1 A realistic example

We are applying these representations to a real problem: We are building a software application named MArSHLAnd [16, 17] using an extant application (OPNET Modeler [18]) and source code library (a genetic programming toolkit) as appliponents. MArSHLAnd stands for MAS + HLA (Multi-Agent System + High Level Architecture [19]). Multi-agent systems [20] represent an interesting problem-solving methodology based on localized information, autonomy, distribution, and emergent behavior. The HLA is an IEEE/DoD software infrastructure for interoperable simulations [21]. MArSHLAnd’s thesis is to harness the synergy between genetic search and distributed simulation to guide automated search for MAS designs with high utility or fitness. Genetic search is inherently parallelizable and HLA simulations are inherently distributed and execute in parallel. Typical genetic algorithm is presented in figure 4.1.

Typical algorithm we want to use OPNET Modeler to evaluate the fitness of MAS designs, instead of coding an ad hoc MAS simulator within the GP toolkit. Unfortunately, the GP toolkit does not export its data type for “individuals” (i.e., tree structures that represent the characteristics of candidate designs). The data structure for an individual must be forcibly exported out of the GP toolkit (via intrusive modifications to GP toolkit code), translated into an equivalent representation that can
be imported into OPNET Modeler and executed. After the OPNET Modeler simulation has executed, fitness data collected from the run must be read from a data file and spliced back into the data structure for the individual.

The GP toolkit features a user-defined exit function named app_eval_fitness, where the user must code a custom implementation to evaluate the fitness of an individual. Our implementation of this function will feature all the messy ad hoc software interfacing code that will export the data structure for an individual, transform it, run OPNET Modeler on it, collect the fitness data and assign them into defined fields of the individual. The algorithm of the above mentioned solution is represented in figure 4.2. The approach suggested above is preferred as the number of new software to be developed is minimal. The appliponents that need to be developed are the app_eval_fitness function, ListToTree.exe, Vb.wsc and update_fitness subprogram.
Figure 4.2 Proposed Algorithm in MArSHLAnd – Original Representation

Figure 4.3 shows the ADD the above problem. This diagram replaces the figure 4.2 and shows the dataflow in more detail. The outer most appliponent is the app_eval_fitness subprogram of GP tool kit, which takes the current individual to be evaluated as a memory uncontrolled input (the individual is the formal parameter for this function) and takes the global variable of type globaldata as interaction-in data.

The first form of transformation is accomplished by simply writing the contents of the individual as a list to a comma-separated file. Due to the absence of XML parsing features in GP, an application “List2Tree” was written in VB.Net [33]. List2Tree application executes reads a flat file specified as command line argument (recall that this is of activation in memory uncontrolled data type) and outputs an XML file representation of the individual (SC Passivation). This XML tree is traversed and transformed in to a Deterministic Finite Automaton (DFA) with well-defined states and transitions. OPNET 9.0 version provided the functionality of importing process models as XML specifications. The document type definition (DTD) defining the
schema/structure of the XML document used by OPNET was well-defined and hence the DFA was written to an XML document which conforms to OPNET’s DTD.

Although OPNET’s user interface supported the import of XML specification through the user interface, the import XML External Module Access (EMA) API shipped with OPNET was buggy and did not work according to specifications. At this stage we had two choices:

1. Transform the DFA.xml to an EMA application program written in C.
2. Use Windows Script (WS) [34] to automate the keystrokes to create the process model and save it as a process model file.

The second approach was chosen as the time involved in coding the solution would be considerably less and as a bug fix or the release of next version of OPNET was anticipated.

Simulation results were obtained by executing the op_runsim.exe appliponent. The fitness information was written to a custom defined format with *.gdf (SU) extension. This file was then parsed in the GP tool kit to update the fitness of the current individual.
CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

Symbols and notations recommended for use in ADD are visually intuitive and are commonly used symbols and icons in the Industry. Hence any new user trying to use this notation can adapt very easily.

A detailed analysis and a rigid framework have been presented in this research. Based on these definitions, it would be easy to develop a CASE tool that would act like a visual programming tool for legacy application integration.

ADD representation satisfies 6 of the 7 rules suggested for the success of any visual representation as suggested by [5]. Summary of the rules along with a description of how ADD satisfies the rule is presented in table 5.1

5.2 Potential application areas of ADD

ADD can be used to demonstrate interaction between middleware technologies and enterprise applications by modeling middleware as an appliponent and the enterprise as another appliponent. The data flowing across these appliponents can be classified in to one of the 16 categories and documented. Documents such as these can help in integrating the enterprise application with other middleware.

The main problem in the case of legacy application is the lack of documentation indicating the data interfaces available and the data type for each type of data. Hence every attempt to use the legacy application requires the programmer to spend time identifying interfaces. ADD could be used to document legacy enterprise
application projects by treating legacy applications as one appliponent with each service provided by the legacy application as a function.

Table 5.1 Summary of Rules Satisfied By ADD

<table>
<thead>
<tr>
<th>Rule Condition</th>
<th>Provisions in ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>The extended class diagrams and the package diagrams can clearly show the overview of the application under consideration.</td>
</tr>
<tr>
<td>Zoom</td>
<td>The amount of details to be shown at any ADD is requirement dependent and can be zoomed in to show the specifics.</td>
</tr>
<tr>
<td>Filter</td>
<td>Functions and attributes in the class diagram are filtered based on context.</td>
</tr>
<tr>
<td>Details-On-Demand</td>
<td>Appliponent can be further decomposed to show the internal data flow.</td>
</tr>
<tr>
<td>Relate</td>
<td>The static relationships between the various appliponents can be seen from the normal UML diagram. ADD also shows the dependencies between an appliponent and other state variables.</td>
</tr>
<tr>
<td>History</td>
<td>Only Applicable for applications</td>
</tr>
<tr>
<td>Extract</td>
<td>ADD can extract a specific portion and treat that sub component as an appliponent and represent data flow.</td>
</tr>
</tbody>
</table>

ADD can be used in the design of distributed object technology (DOT) based components and tools by modeling the component itself as an appliponent and documenting the data interfaces to the component.

40
5.3 Future Work

Appliponent Diagram currently can be used extensively to show flow of data in sequential manner. In future, the notations have to be improved to incorporate branching, looping, recursion, etc. Current version of appliponent diagrams does not have a way to categorize exception handling or raising in our 16 different kinds of interface data or the control flow during the same.

The modeling at this stage does not have notations to show parallel, asynchronous message passing or interactions between appliponents.

We want to improve the precision and accuracy of our definitions, especially with regard to what comprises activation inputs, passivation outputs, interaction inputs, and interaction outputs.

The first approach would be to categorize these data according to the data flow anomalies they represent. E.g., activation inputs could be defined to be those data that for any path through the appliponent’s control flow graph (CFG), the first “action” on those data is a reference. Passivation outputs could be defined as those data that for any path through the appliponent’s CFG, the last “action” on those data is an assignment. Interaction inputs and interaction outputs could be defined as those data for which both the above data flow anomalies exist. Definitions such as these are problematic for interaction inputs and interaction outputs, because the definitions fail to always categorize class attributes as interaction inputs and interaction outputs. Recall class attributes are defined to store object internal state. We could make exceptions to the
above definitions to force attributes and hidden internal state variables to be interaction inputs or interaction outputs.

A second approach would employ not just a CFG for each appliponent, but a Class Flow Graph [22] for the class containing the related appliponents. A Class Flow Graph is a composite of the state transition graph (STG) for the class as a whole and the control flow graphs (CFGs) for each method defined in the class. In a Class Flow Graph, each transition defined in the class’s STG is replaced with an instance of the CFG of all methods called along that transition. This approach most likely will not “correctly” categorize keyboard and mouse as interaction inputs and video as interaction outputs.

A third approach would be to consider those inputs and outputs that are repeatedly read from and written to, respectively. It is not yet clear how to come up with a precise definition of “repeatedly”.

A CASE tool needs to be developed with a parser that can ease creating new appliponents, provide a way to type check different data inputs and generate glue code implementations for the ADDs.

Although accessibility of data has been classified to fit in to two discrete domains, in reality, data accessibility is a continuous domain. Hence the current data accessibility model needs to be improved to categorize data by measuring in logarithmic scale, the amount of human effort or some other utility functions need to be incorporated.
The issues with data aliasing need to be addressed in the model to differentiate between data passed as reference and data passed as value or both.

Utility of Appliponent diagrams needs to be compared with other traditional modeling methods for a more potent example.

The concept of appliponents causes the engineer to think in terms of data flow and applicative programming style. We envision a tool in which the programmer selects source code within an editor, invokes a data flow analysis tool on the selected source code, and automatically discovers the implicit inputs and outputs of the selected source code. Such a tool could be used during program modification to ensure that implicit inputs and outputs change only in desired ways. Another use of such a tool would allow a programmer to select source code text and automatically generate a fully-parameterized subprogram and invocation to replace the selected code. This could be useful for re-modularizing ugly source code.
REFERENCES


46
[28] Visual Application Builder - white paper about infinity project

[29] LabVIEW home page, Tutorial and References
   http://zone.ni.com/devzone/conceptd.nsf/webmain/F34045D2CC5357F486256D3400648C0F?opendocument

[30] Karel's Programming Environment home page,
   http://www.cis.ufl.edu/~jnw/Karel/gui_picture.gif

[31] LEGO mind storm software screen shot
   http://sensorscity.marshall.edu/CITY/pages/vi/ PictureOfAprilProgram.jpg

    Wright, and T. Xu, BizTalk Server: The Complete Reference,

[33] Visual Basic .net home page, References and Tutorial,

[34] Windows Scripting home page, technology and references,

[35] The International Standard for Electronic Data Interchange, United Nations
    EDI For Administration, Commerce and Transport,
    http://www.unedifact.com/edi/edi_0.htm

[36] MSW Logo – A modern LOGO version Screen Shot,

The author was born in Tamil Nadu, India, November 15, 1979. He graduated from University of Madras, India, in 2000 with a Bachelor’s degree in Civil Engineering. He came with a Dean’s fellowship to the The University of Texas at Arlington in 2000 to pursue Master of Science in Civil Engineering. At this institution he worked as a research assistant from spring 2001 to spring 2002 in the Department of Civil and Environmental Engineering. Mohan received his Master of Science in Civil Engineering from The University of Texas at Arlington in August 2002. He also received scholarship for his work in geotechnical instrumentation. He continued to work as a graduate research assistant in Department of Electrical Engineering, Department of Computer Science and Engineering and in the Counseling and Advising Center. He received his Master of Science in Computer Science and Engineering from The University of Texas at Arlington in May 2004. During this time he was awarded a merit scholarship from the Department of Computer Science and Engineering.