Ad Hoc Software Interfacing: Enterprise Application Integration (EAI) when Middleware is Overkill

Arthur A. Reyes, reyes@cse.uta.edu
José R. Espino, espino@cse.uta.edu
Vijai Mohan, vmohan@cse.uta.edu
Monica Nadkar, nadkar@cse.uta.edu


This report was also submitted to the 27th Annual International Computer Software and Application Conference (COMPSAC 2003) Workshop on Architectures for Complex Application Integration (WACAI), Dallas, Texas, USA, November 3-6, 2003.
Ad Hoc Software Interfacing: Enterprise Application Integration (EAI) when Middleware is Overkill

Arthur A. Reyes  
Assistant Professor  
reyes@cse.uta.edu

José R. Espino  
Graduate Student  
espino@cse.uta.edu

Vijai Mohan  
Graduate Student  
vmohan@cse.uta.edu

Monica Nadkar  
Graduate Student  
nadkar@cse.uta.edu

Dept. of Computer Science & Engineering  
University of Texas at Arlington  
Arlington, TX, 76019-0015 U.S.A.

Abstract

Enterprise application integration (EAI) is cooperation of disparate systems and components to implement business rules in a distributed environment. “Systems and components” can be computer-aided design (CAD) or software engineering (CASE) tools, enterprise databases, COTS applications, or in-house software. Ad hoc software interfacing (AHSI) is a special kind of EAI. A tradeoff analysis classifies an EAI problem as an AHSI problem when middleware solutions are seen as heavy-handed. I.e., the planned EAI is not expected to become broad enough to justify the generality of a middleware solution or the client is unwilling to pay for a unified data model. AHSI seeks to “wire” extant software applications as components in new, larger software applications. We call applications-as-components “appliponents”. AHSI seeks to minimize appliponent modification to the greatest extent possible. We demonstrate solutions to AHSI problems using XML toolkits, domain-specific language toolkits, and Microsoft BizTalk Server.

1. Introduction

“Software interoperability” is “the ability of two or more systems or components to exchange information and to use the information that has been exchanged.” [7] Software interoperability has made significant advances for popular execution platforms, such as Windows and its related products. Middleware, such as CORBA [9] and .NET [13] represent detailed, technical conventions that allow independently-developed software components to interoperate.

Not every software product is CORBA-compliant, .NET-compliant, or (other middleware)-compliant. Many software applications developed for in-house use slowly accrete features over a period of years. Such software becomes “legacy”. Even though such applications can become difficult to maintain, owning organizations are loathe replacing them with modern, pre-packaged software products with similar features. This can happen because of the large “investment” made in evolving such applications to date and the uncertainty involved in switching to a pre-packaged solution.

Occasionally the need arises to make the legacy system interoperate with another system (possibly also a legacy system). This is an example of enterprise application integration (EAI). “Enterprise application integration may be defined as the cooperation of disparate systems and components to implement business rules in a distributed environment.” [8] EAI accommodates a number of technologies, such as RPC, object-oriented middleware (CORBA, COM, .NET), and message-oriented middleware (MOM) [10].

At the beginning of such a project, software engineers need to conduct a trade-off study to evaluate the costs of the various interoperability solutions. Consider the following contrived example.

1.1. A Contrived, but Illustrative Legacy System Interoperability Trade-Off Analysis

Suppose you need to interface legacy system A with legacy system B. Suppose also A and B are “stovepipe” systems—vertically but not horizontally integrated. A supports feature X only. B supports feature Y only. Suppose A does not import or export data, but it reads and writes data files associated with feature X. Likewise suppose B does not import or export data, but it reads and writes data files associated with feature Y. A and B can reside on different hosts. Suppose we need to support features X and Y and that the X and Y features use each others’ data. There are several approaches to providing the combined X+Y feature:

- Define ad hoc data translations between A and B.
- Extend A with feature Y. Do not use B.
- Extend B with feature X. Do not use A.
- Make A & B .NET-compliant.
- Make A & B COM-compliant.
- Make A & B CORBA-compliant.
- Make A & B HLA-compliant.
- Build features X & Y from scratch.
Make A & B interchange streamed data in a canonical format.
Import data from A to B, manually update A from B.
Import data from B to A, manually update B from A.
Run A, manually enter the X data into B, run B,
manually enter the Y data into A, etc.

We can carry out an informal, “back-of-the-envelope” cost analysis for each of these 12 solutions. Such an analysis can be performed to discover decision parameters and check our intuition and locate omissions in our planning. Table 1 recapitulates our assumptions.

Table 1 Assumptions for the Legacy System Interoperability Trade-Off Analysis

<table>
<thead>
<tr>
<th>assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A exports no data</td>
</tr>
<tr>
<td>A imports no data</td>
</tr>
<tr>
<td>A supports feature X</td>
</tr>
<tr>
<td>B exports no data</td>
</tr>
<tr>
<td>B imports no data</td>
</tr>
<tr>
<td>B supports feature Y</td>
</tr>
<tr>
<td>Features X &amp; Y must exchange data</td>
</tr>
<tr>
<td>The composite application supports features X &amp; Y</td>
</tr>
</tbody>
</table>

Table 2 documents our cost categories (each number equals the prior number raised to 1.25). VS is very small, S is small, M is medium, L is large, and VL is very large. The units of cost are not important. Exponentially-spaced categories such as these are exactly what you get if you perform a log-normal transformation on your historical data. You would perform such a transformation on your historical data when the standard deviation is greater than ¼ the mean (i.e., this can happen when your historical dataset is small). The log-normal transformation ensures that the resulting categories are non-negative. This is a practice of the Personal Software Process [6].

Table 2 Cost Categories for the Legacy System Interoperability Trade-Off Analysis

<table>
<thead>
<tr>
<th>Cost Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>L</td>
</tr>
<tr>
<td>VL</td>
</tr>
</tbody>
</table>

Table 4 summarizes the analysis of development costs and costs that recur on a per-run basis for each solution. Each row represents one of the 12 solutions options described above. We assign all cost values from the cost categories above. Of course we can assign any cost category we please; we allow our intuition to guide our choices. Hopefully the resulting analysis will bring us to a more accurate understanding of the trade-off qualities.

“Cost of Mods to A” is the development cost associated with modifying A. “Costs of Mods to B” is the development cost associated with modifying B. “Other Development Costs” are the costs of developing the “main” program that calls A and B, etc. We are going to plot cost versus the number of times the resulting solution is run. Thus Total Development Cost is the cost accrued at the first run (i.e., the y-intercept). Run Cost is the cost of each run of the resulting solution. “Cost @ Run # 1” is the sample cost at the first run of the resulting solution. “Cost @ Run # 100” is the sample cost at the 100th run of the resulting solution. The rows are sorted in ascending order according to their cost at the 100th run. Because our trade-off assumes a linear relationship between cost and number of runs, we only need two points to plot the function.

Figure 1 graphs cumulative cost as a function of number of runs for the 12 solutions. Three solutions are far too costly if the solution is to be run many times. They are the solutions that require the user manually enter data into A, B, or both.
2. Ad Hoc Software Interfacing

Ad hoc software interfacing (AHSI) is a subset of EAI where the task of developing middleware wrappers is not cost-effective for the system at hand and its expected future needs. We want to identify effective and maintainable technologies and methods that will allow a large class of software applications to be used as components in new, larger software applications. We refer to such applications-as-components as "appliponents". Moreover AHSI is applicable when neither the A nor B appliponents support a middleware technology, there is no need for a unified data model, or only a small fraction of each appliponent’s data must be interchanged. AHSI proceeds under the assumption that it is preferable to solve an EAI problem by composing (often cobbling) extant appliponents together, even if the interfaces the appliponents export are empty, incomplete, or inconvenient at present; rather than building new software components with cleaner interfaces.

2.1. Framework

AHSI decomposes an EAI problem into a graph of point-to-point data transformations. Data at each end of each transformation have a type. Table 4 describes high-level categories for data. Each cell gives examples of data in that category. Data can exist in streams (including files) or in memory. Adequate, programmatic access to data can be predefined or not.

<table>
<thead>
<tr>
<th>Table 4: Type Categories in Ad Hoc Software Interfacing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Type</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Predefined</td>
</tr>
<tr>
<td>~Predefined</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 5 recapitulates Table 4, but shows a Roman numeral for each quadrant, much in the style of dividing the Cartesian coordinate space into four quadrants.
Quadrant IV is arguably the least desirable, because reading or writing such data requires intrusive modification to the appliponent that owns such data. Quadrant I is arguably the most desirable.

Table 5 Quadrants in the Type Space

<table>
<thead>
<tr>
<th>Access</th>
<th>data are in files/streams</th>
<th>data are in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>predefined</td>
<td>II</td>
<td></td>
</tr>
<tr>
<td>~predefined</td>
<td>III</td>
<td>IV</td>
</tr>
</tbody>
</table>

If you rotate Table 5 90 degree counterclockwise, you can think of the “Access Predefined” column as a protocol stack and the “Access ~Predefined” column as another protocol stack. As indicated by the arrows, if you need to move data between and quadrant I type A and a quadrant IV type Z where A and Z reside in different hosts, you will need pass the data down a protocol stack to the stream level, transmit the data at that level, and move the data up the other protocol stack. If you need to translate the data along the way, you can do it at the stream level via scripting languages like Perl [11], stream editing languages like sed [4]. If the data source is an XML stream, you can perform a context-free translation into another text stream via XML Stylesheet Language (XSL) [29]. You can perform less-restricted translations using XML Stylesheet Language for Transformations (XSLT) [28].

2.2. Contrived Example of AHSI

Figure 3 demonstrates the phases of AHSI. Suppose we need an application built from a number of appliponents. Each appliponent uses a collection of types, some of which are named (e.g., those exported by one of the appliponent’s application programming interfaces (APIs)) and some of which are not named (e.g., input and output files). Types are shown as nodes; functions, procedures, methods, etc. that operate on types are shown as hyper-arcs (i.e., arcs that can map a set of nodes to a set of nodes). This notation is inspired by the “ADJ” diagrams [5] used to graphically specify the signatures of abstract data types. This is an applicative notation, thus side-effects of functions, procedures, methods, etc. have to be explicitly shown at the heads of hyper-arcs. Later we will show these as class diagrams, where functions, procedures, methods, etc. are shown as unidirectional associations between “classes”. Functions, procedures, methods, etc. that take more than one argument or return more than one value (or that have side effects on several objects) are shown as n-ary associations (diamond symbol). It is sometimes helpful to partition the nodes of such graphs according to their corresponding type quadrant.

Figure 3 Graphical Depiction of Ad Hoc Software Interfacing

In Phase 1, we discover that we need to transform data of type A to corresponding data of type Z. Thus the goal is to draw a directed (hyper-)graph from A to Z where the new nodes and (hyper-)arcs of the graph represent extant types, classes, functions, procedures, methods, etc. given the appliponents of interest. In Phase 2, we discover that there exists a type B and a promising function $f$ that maps A objects into B objects. If A is an object class and B is a stream or file type, $f$ could represent a function that serializes an A object to a stream for persistent storage. In Phase 3, we discover that there exist types C and D and promising functions $g$ and $h$, but $g$ and $h$ deal with distinct parts of the B object of interest to us. In Phase 4, we discover a function (or more likely develop a new function from scratch) $i$ that maps a (C, D) pair of objects into a Z object.

2.3. Developing Arcs

There are 16 combinations of data transfer between the four type quadrants. Table 6 describes what must be done to accomplish each kind of mapping. The table uses the word “space” as follows: if the A→B map crosses from “access predefined” to “access not predefined” or vice versa, “same space” means same computer host; if the A→B map does not cross from
Suppose you need to map streamed A data to streamed B data, where there is no extant mapping function or convenient interoperability mechanism, but A data and B data reside on the same file system. If you know the concrete syntax of A data, the concrete syntax of B data, and the semantics of the mapping, you can generate a mapping function via syntax-directed translation tools such as yacc [1]. The basic mapping process is shown in Figure 4. “A” represents a data stream in the concrete syntax of type A. “B” represents a data stream in the concrete syntax of type B. “parser” represents a parser generated by a tool such as yacc. “A (abstracted)” represents the parse tree or abstract syntax tree (AST) of the A stream. “Abstract A → B translator” represents the abstract mapping function generated from the syntax-directed translation rules. “B (abstracted)” represents the parse tree or AST of the B stream. “printer” represents the print/write function generated by a tool such as yacc.

```plaintext
                    A (abstracted)    B (abstracted)
    A "parser"   ←→    B "printer"
         A           ->         B

Figure 4: The Parse; Transform; Print Cycle
```

Software toolkits exist to generate the parser, printer, and possibly the translator from formal specifications of the A and B languages being mapped. Figure 5 enhances Figure 4 with this information. The user of such a toolkit writes a formal specification of the concrete and abstract syntax of the A and B language and a “formal specification” of the mapping from A ASTs to B ASTs—although the latter usually takes the form of a concise, often applicative, computer program.

```plaintext
                    A (abstracted)    B (abstracted)
    You formally specify A’s "syntax".    You formally specify B’s "syntax".
         A           ->         B

Figure 5: Specified vs. Generated Parts of the “Parse; Transform; Print” Cycle
```

There are only a handful of different processes at work here: the “parse;transform;print” cycle, file transfers, calls to API functions, calls to middleware functions, reading, and writing. The only one we will expand upon is the “parse;transform;print” cycle.

```
<table>
<thead>
<tr>
<th>Case</th>
<th>A in Quad</th>
<th>B in Quad</th>
<th>A, B in same space</th>
<th>A, B in different space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I</td>
<td>I</td>
<td>compose call API fns</td>
<td>compose call M fns</td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>I</td>
<td>API print</td>
<td>FTP</td>
</tr>
<tr>
<td>3</td>
<td>I</td>
<td>I</td>
<td>API print; FTP</td>
<td>custom B read</td>
</tr>
<tr>
<td>4</td>
<td>I</td>
<td>I</td>
<td>API print; FTP</td>
<td>custom B read</td>
</tr>
<tr>
<td>5</td>
<td>I</td>
<td>I</td>
<td>API print</td>
<td>FTP</td>
</tr>
<tr>
<td>6</td>
<td>I</td>
<td>I</td>
<td>API print</td>
<td>FTP</td>
</tr>
<tr>
<td>7</td>
<td>I</td>
<td>I</td>
<td>custom script or PTP</td>
<td>FTP</td>
</tr>
<tr>
<td>8</td>
<td>I</td>
<td>I</td>
<td>custom script or PTP</td>
<td>custom B read</td>
</tr>
<tr>
<td>9</td>
<td>I</td>
<td>I</td>
<td>reverse of case 3 &quot;same&quot;</td>
<td>reverse of case 3 &quot;different&quot;</td>
</tr>
<tr>
<td>10</td>
<td>I</td>
<td>I</td>
<td>reverse of case 7 &quot;same&quot;</td>
<td>reverse of case 7 &quot;different&quot;</td>
</tr>
<tr>
<td>11</td>
<td>I</td>
<td>I</td>
<td>custom script or PTP</td>
<td>FTP</td>
</tr>
<tr>
<td>12</td>
<td>I</td>
<td>I</td>
<td>custom script or PTP</td>
<td>FTP; custom B read</td>
</tr>
<tr>
<td>13</td>
<td>I</td>
<td>I</td>
<td>reverse of case 4 &quot;same&quot;</td>
<td>reverse of case 4 &quot;different&quot;</td>
</tr>
<tr>
<td>14</td>
<td>I</td>
<td>I</td>
<td>reverse of case 8 &quot;same&quot;</td>
<td>reverse of case 8 &quot;different&quot;</td>
</tr>
<tr>
<td>15</td>
<td>I</td>
<td>I</td>
<td>custom A write; FTP;</td>
<td>custom B read</td>
</tr>
<tr>
<td>16</td>
<td>I</td>
<td>I</td>
<td>custom A write; FTP;</td>
<td>custom B read</td>
</tr>
</tbody>
</table>
```

Table 7 Legend for the Table Above

<table>
<thead>
<tr>
<th>term</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>memory, access predefined</td>
</tr>
<tr>
<td>II</td>
<td>stream, access predefined</td>
</tr>
<tr>
<td>III</td>
<td>stream, access ~ predefined</td>
</tr>
<tr>
<td>IV</td>
<td>memory, access ~ predefined</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AST</td>
<td>abstract syntax tree</td>
</tr>
<tr>
<td>fns</td>
<td>functions</td>
</tr>
<tr>
<td>FTP</td>
<td>file transfer</td>
</tr>
<tr>
<td>M</td>
<td>Middleware</td>
</tr>
<tr>
<td>parse</td>
<td>maps a stream into an AST</td>
</tr>
<tr>
<td>print</td>
<td>maps an AST into a stream</td>
</tr>
<tr>
<td>PTP</td>
<td>parse;transform;print</td>
</tr>
<tr>
<td>read</td>
<td>maps a stream into a memory object</td>
</tr>
<tr>
<td>script</td>
<td>maps a stream into another stream</td>
</tr>
<tr>
<td>transform</td>
<td>maps an AST into another AST</td>
</tr>
<tr>
<td>write</td>
<td>maps a memory object to a stream</td>
</tr>
</tbody>
</table>

X;Y X is performed before Y

Table 6 Mapping Between Each Pair of Type Quadrants
Domain-Specific Language (DSL) toolkits have been developed since the 1970s as enhancements to parser-generators. The Related Work section reviews four such toolkits. More recently, XML toolkits have come to duplicate and surpass many of the features of DSL toolkits.

3. A Realistic Example

We are applying these concepts to a real problem: We are building a software application named MArSHLAnd [22, 23] and source code library (a genetic programming toolkit) as appliponents. MArSHLAnd stands for MAS + HLA (Multi-Agent System + High Level Architecture). Multi-agent systems [24] 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 [3]. 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.

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 a C source code that makes calls to OPNET Modeler’s External Model Access (EMA) library, compiled, linked, and executed. After the OPNET Modeler simulation has executed, fitness data collected from the run must be read from a 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.

Figure 6 shows the AHSI graph of types and functions. The rectangle at the left center represents the GP toolkit’s data type for individuals. This rectangle is both the A type and the Z type. The rectangles at the top center and top right of the graph represent Erlang [16] types that we used and extended. We used Erlang to write executable models of functions that transform a preorder list of nodes back into the original tree, decode the tree into a graph representing a state transition machine, and emit an OPNET Modeler EMA C source code application. Under the AHSI paradigm, there is no need to stop using the Erlang executable model code: just interface it into the solution. The nodes at the bottom center and bottom right represent types associated with various OPNET Modeler files. The node at the left bottom represents an ad hoc file format in which to initially write out the contents of the GP toolkit individual from memory. The diagram shows the four quadrants of the type space arranged from top to bottom: I, IV, II, III. Nodes are placed in their corresponding lane. Erlang does not feature named types, so our emerging type system for AHSI will need to account for this phenomenon as well.

![Figure 6 Ad Hoc Software Interfacing for MArSHLAnd](image)

The AHSI solution for MArSHLAnd described above is being re-implemented using Microsoft BizTalk Server. We hope the BizTalk implementation will be easier to specify and maintain than the solution described above.

4. Microsoft BizTalk Server

BizTalk Server [18] is an EAI server that is built on top of the Microsoft XML Toolkit and several other technologies (such as Microsoft COM). BizTalk enables dissimilar applications to interoperate by passing XML data streams to each other, even if the dissimilar applications do not input or output XML. With the BizTalk Editor, you specify the relevant data formats for each dissimilar application. If the application does not input or output XML (i.e., the application uses a “native format”), the BizTalk Editor can generate scripts that transform native input to XML and vice versa. Data formats, specified using the BizTalk Editor, are represented as generated XML Schema Definition (XSD) files. Data formats are the syntax of simple
languages (i.e., fixed-width or delimited flat files). The BizTalk editor can also generate “instances” of data formats you define so you can check whether you are defining the data format correctly. Figure 7 shows a screenshot of the BizTalk Editor.

Figure 7 BizTalk Editor

4.1. Schemas from the MArSHLand Example

In our re-implementation of MArSHLand via BizTalk Server, every node (i.e., type, class) in the class diagram of Figure 6 will be represented by at least one XSD. The XSD specifies the syntax for that type of data. Every edge in Figure 6 (i.e., procedure, function, method, etc.) extending between two nodes needs an XSLT to specify the transformation.

To illustrate this point we now describe schemas for “IndFlatDat” and “CJCappControlProcTree”. IndFlatDat is a native format representing a preorder list of nodes in the individual’s genetic chromosome tree. CJCappControlProcTree is an XML format representing the individual’s genetic chromosome tree\(^1\). IndFlatDat will become a source schema. CJCappControlProcTree will become a destination schema. The process of creating the specification for the preorder list can be viewed as a method of specifying rules to parse the flat file that was written out from the memory of the GP toolkit kernel. Furthermore, this XSD also provides a way of printing a XML file to flat, native data.

We have the destination specification called “CJCappControlProcTree”. In the screenshot shown below, the pane on the left shows the hierarchy of the specification, while the pane on the right displays some properties of “Node”. The left pane has three fields called “Id”, “Parent”, and “Value” and a record of type “Node” which is created as a descendant to itself. This is called Cyclical Reference. The “Id” is the unique identifier for each node.” “ParentID” signifies the parent of every node and “Value” is the degree of every node. We have a record of type “Node” because a node itself may be composed of one or more children.

\(^1\) Such genetic chromosome trees are structured in the style of [Brave].

The BizTalk Mapper allows you to graphically specify transformations between data formats. Graphical specifications are stored as XML Stylesheet Language for Transformations (XSLT) files. Figure 9 shows a screenshot of the BizTalk Mapper. Mappings between elements and attributes of two schemata can be complex. “Functoids” are nodes in the mapping that represent arbitrary functions that generate destination schema element or attribute values from a collection source schema elements and attributes.

Figure 8 Schema of CJCappControlProcTree

Figure 9 BizTalk Mapper

The BizTalk Editor and Mapper thus capture the user’s formal specifications of the source and destination languages (schemata) and the semantic transformation of the source language into the destination language. From these specifications, a parser, transformation engine, and a printer can be generated automatically.
4.2. Mapping in the MArSHLAnd

Example

The screenshot below shows the mapping between the two schemas described above. We have used various functoids to enable the complex transformation from the IndFlatDat schema to recursive CJCappControlProcTree schema. As this transformation is inherently complex, scripting functoids were created using Visual Basic. Every node from the source is set to a level of hierarchy in the destination based on simple mathematical calculations utilizing the nodes attributes and the state of the transformation.

![Figure 10 IndFlatDat → CJCappControlProcTree Mapping](image)

The BizTalk Orchestration Designer allows parse;transform;print cycles to appear anywhere needed in a workflow. Figure 11 shows the BizTalk Orchestration Designer. The workflows you specify with the Orchestration Designer get data from a number of dissimilar enterprise applications, make decisions about what to do based on the values of the data, and push new data into a number of dissimilar enterprise applications. Workflows are essentially Activity Diagrams with “ports” and “implementation shapes”. Implementation shapes represent enterprise applications, which can communicate with BizTalk Server via COM, Windows Scripts, SMTP, or HTTP. Ports represent the places where the schema-to-schema transformations occur.

![Figure 11 BizTalk Orchestration Designer](image)

4.3. Orchestrating MArSHLAnd

The screen shot below shows the orchestration design for MArSHLAnd. The ‘Create Individuals’ action represents the population of individuals composed of genetic program (GP) chromosome trees. A program we intrusively added to the GP toolkit kernel writes the data structure of an individual (a tree represented as a preorder list of nodes) to a flat file (of “type” IndFlatDat). The IndFlatDat file is read into an XML stream. The XML stream is transformed several times and a flat, destination document is given as input to OPNET Modeler. OPNET incorporates the design, executes the simulation, and gathers statistics. Fitness data from the simulation is fed back to the GP process though the read_fitness function. This process is repeated until an individual with required properties is discovered by the GP toolkit.

![Figure 12 Orchestration Design for MArSHLAnd](image)

5. Related Work

As related work, we considered “software design patterns”, DSL and XML toolkits, and software generators built on top of DSL and XML toolkits.

5.1. Software Design Patterns

Software design patterns are presented in the seminal work of [31]. In the words of the authors, “each pattern systematically names, explains, and evaluates and important and recurring design in object-oriented systems.” A large number of design patterns beyond those originally proposed by the “gang of four” (GoF) (e.g., Memento, façade, bridge, etc.) have been developed (see [32]). Included in the GoF design patterns are the Adapter, Proxy, and Bridge which are of particular interest because they provide another way of
thinking about the AHSI problem. Consider Table 4 (previously presented in section 2.1) which depicts four quadrants and Figure 13.

![Figure 13 Quadrants Occupied by Three Selected GoF Design Patterns](image)

The four quadrants of Table 4 have been superimposed over three design patterns (each design pattern is surrounded by a box). Included in Figure 13 are Adapter, Bridge, and Proxy, occupying the top, middle, and bottom positions, respectively. The Adapter design pattern is used when a need to resolve incompatible interfaces arises where we may have a software product that needs to support third-party services (Web Service, authorization services, etc.). Clearly, it is not possible to modify the incompatible interfaces. The Target class lies in the (Memory, Access Predefined) quadrant (the class instance exists in memory and the methods are the one we want) and the Adaptee class is in the (Memory, ~Access Predefined) quadrant (the class is in memory, but the access is not available because the interface is incompatible). The Adapter class may provide additional functionality to the Adaptee and makes its interface compatible. This is analogous to starting in quadrant I, moving through quadrants II and III, and finishing in quadrant IV.

The Bridge is used to help separate abstraction from implementation. According to [31] the design pattern Bridge should be used when we “want to avoid permanent binding between an abstraction and its implementation.” This is an attractive feature because in the AHSI problem, the components may change.

The Proxy design pattern is used as a surrogate or placeholder—it is a type of intermediary between the subject and the real-subject. In AHSI, there may be several to many independent, processes (different address spaces). The design pattern Remote Proxy would be used for inter-process communication and would be analogous to the AHSI mapping data (from stream to stream) from process A to process B.

### 5.2. DSL and XML Toolkits and Generators

Table 8 summarizes DSL or XML toolkits that can generate AHSI solutions from specifications and software generators built from DSL or XML toolkits. Table 8 features the following columns:

- **Application Domain**: What is the targeted application domain of the toolkit or the example software generator built from a DSL or XML toolkit?
- **Source Stream Representation**: What is the input to the parse;transform;print cycle?
- **Parse Process**: How is the parse process enacted?
- **AST Representation**: How is the resulting AST represented?
- **Transform Process**: How is the transformation process enacted?
- **Print Process**: How is the print process enacted?
- **Destination Stream Representation**: What is output of the parse;transform;print cycle?
- **Number of parse;transform;print Cycles Typically Supported**: Can you use the referenced system to aggregate parse;transform;print cycles together in a complex workflow?
- **How is DSL Syntax Specified?**
- **How are Transformations Specified?**
- **How are Workflows Specified?**
- **What is generated?** Exactly what software artifacts are generated?
- **How wraps dissimilar applications?** Does the referenced system have any support for “wrapping” dissimilar software applications?

### 5.3. DSL Toolkits and Example Generators

DSL methods and toolkits are oriented toward generating DSL processing tools from syntactic and semantic specifications of the language to be processed. DSL processing tools can be syntax checkers, type checkers, evaluators, translators, and compilers. Syntactic specifications take the form of grammars. Semantic specifications take many forms, depending on the toolkit and the requirements for the DSL under development. DSL toolkits seem to be used mostly for prototyping new or domain-specific programming languages. We survey ASD+SDF [14], Erlang YECC [28], JTS Jakarta Tool Suite [18], Reasoning SDK [2, 10] and example generator Bell Labs Network Feature Service Generator [15].
5.4. XML Toolkits

XML (eXtensible Markup Language) is a simple, fully-bracketed, easy-to-transmit, pre-parsed, extensible format for representing structured data. There is a family of XML languages:

- **DTD (Data Type Definition)** is a grammar specification language. It specifies the valid syntax for XML “business documents” that use a defined set of “elements” and “attributes”. XML business documents that conform to such a grammar are said to be “valid”. DTD is being phased out because it is actually not itself an XML language. Its replacement is XSD, described next.

- **XSD (XML schema definition)** language fulfills the needs of DTD. XSD is itself an XML language.

- **XSL (XML Stylesheet Language)** is an XML language for specifying context-free translations on XML business documents. XSL can be used to translate an XML business document into a corresponding HTML (or VRML, etc.) business document according to formatting rules you define.

- **XSLT (XML Stylesheet Language for Transformations)** is an XML language for specifying arbitrary translations on XML business documents.

- **XPath** is a language for specifying navigation paths through XML business documents.

- **XML DOM (Document Object Model)** is a specification for a library of functions to read (i.e., parse), navigate, query, modify, apply style sheets and transformations, and write (i.e., print) XML business documents.

Allow yourself to think of a DTD or XSD as defining the syntax of a DSL. Calling the DOM reader function “parses” the XML business document into a corresponding “DOM tree” object (i.e., parse tree) in memory. Calling the DOM writer function “prints” the DOM tree object into a stream. Calling the DOM transform function applies an XSL or XSLT specification to a DOM tree object.

XSL and XSLT specifications can be written in a declarative or functional/recursive style.

In short, you could use the XML languages and the DOM to provide almost all of the features available in DSL toolkits. One DSL toolkit feature they do not provide is the ability to start with a non-XML business document as input (e.g., flat file, program file). Thus for each kind of non-XML file type, you must write a translator script that takes the flat file, parses it, and generates its corresponding XML business document. As we will see later, some EAI products generate such translators.

5.5. XML Applications in EAI

5.5.1. TIGRA

TIGRA [20] is a toolkit used for EAI (specifically for financial transactions). It supports transfer of data between applications that were not intended to operate by using a family of XML languages to represent data as well as CORBA Event Notification services for transport of the data. The toolkit is divided into three conceptual components: input adapters that convert the output of heterogeneous front office components first into the Fix protocol (FixML) [26] and FpML [27] and then XML, a Router component responsible for routing the output of each input adapter to the appropriate middle office and back office components, and output adapters that convert the messages into the input required by the target interface.

Consider the typical flow of information through TIGRA. The input to TIGRA consists of flat files (files with data fields delimited by commas or other separators). Application-specific DTDs are used to convert the output files into application-specific mark-up (application-specific XML) files followed by application of XSL and scripting languages (VBscript and/or JavaScript) into other XML languages (FixML or FpML where appropriate). The transformations take place in the input adapters. Once the transformations by the input adapter are complete, the XML message is sent to the Router where routing is accomplished by subscription of suppliers (output adapters) and consumers (input adapters). The target input adapters transform the XML message into the target interface representation via XSLT.

TIGRA claims success based on the number of interfaces it can integrate. Before TIGRA, one interface required “between three and ten person years of effort and took between one and three years” [20]. After using TIGRA, one bank integrated 13 interfaces in one year. TIGRA is designed specifically for financial software only.

5.5.2. GAIL

GAIL [17] is applicable to the B2B problem domain (the creation of an n-tier solution using applications located in different businesses) and is based on the Model Driven Architecture (MDA) defined in [27]. Application integration in GAIL consists of five steps: creation of a UML model that represents the classes that are to be created de novo, specification of an architecture (a platform independent model), mapping of the architecture to a platform specific model, implementation of the architecture using parameterized code templates, and code generation.
GAIL is capable of incorporating design patterns (i.e. Memento) and generating the interface code for the different layers of the specified architecture. It can only provide support for object oriented languages and does not generalize into other problem domains (i.e. simulation).
### Table 8 Summary of Related Work

<table>
<thead>
<tr>
<th>Related Work</th>
<th>Applications Domain</th>
<th>Source Code High?</th>
<th>DSL</th>
<th>XML</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASDF+SDF [14]</strong></td>
<td>Education, Research</td>
<td>Yes</td>
<td>Yes</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td><strong>Erlang YECC [26]</strong></td>
<td>Compiler Generation</td>
<td>Yes</td>
<td>Yes</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td><strong>JTS Jakarta Tool Suite [18]</strong></td>
<td>Research</td>
<td>No</td>
<td>No</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td><strong>Reasoning SDK [2, 19]</strong></td>
<td>Program Transformation, Research</td>
<td>Yes</td>
<td>Yes</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td><strong>Boo Labs [15]</strong></td>
<td>Network Feature Services (NFS), Architecture source code generation</td>
<td>Yes</td>
<td>Yes</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td><strong>MXML Toolkit [19]</strong></td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td><strong>BizTalk [6]</strong></td>
<td>EAI &amp; ESB</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td><strong>TIGRA [20]</strong></td>
<td>financial EAI</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td><strong>GAML [17]</strong></td>
<td>generate code to interface between layers of ESB systems</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
</tr>
</tbody>
</table>

**DSL Toolkits**
- **ASDF+SDF**
  - ASDF text of user-defined DSL
  - call the generated parser
  - user-defined ASDF
  - user-defined ASDF equations are rewrite rules
  - parser
  - parser

- **Erlang YECC**
  - ASDF text of user-defined DSL
  - call the generated parser
  - Erlang text
  - Erlang functions
  - parser
  - parser

- **JTS Jakarta Tool Suite**
  - ASDF text of user-defined DSL
  - call the generated parser
  - Java text
  - Java functions
  - parser
  - parser

- **Reasoning SDK**
  - ASDF text of Ada, C, FORTRAN, user-defined DSL
  - call the generated parser
  - Refine text
  - Refine rules, etc.
  - parser
  - parser

- **Boo Labs**
  - ASDF text of software "view" specification
  - process is not described in the paper
  - process is not described in the paper
  - process is not described in the paper
  - process is not described in the paper

- **MXML Toolkit**
  - XML stream
  - DOM document in memory
  - DOM document in memory
  - XML stream
  - XML stream
  - any

- **BizTalk**
  - XML stream or "native instances"
  - DOM document in memory
  - DOM document in memory
  - XML stream or "native instances"
  - any

**XML Toolkits**
- **ASDF+SDF**
  - XML schema definition file XSD
  - XML stream
  - any

- **Erlang YECC**
  - XML schema definition file XML
  - XML stream
  - XML stream

- **JTS Jakarta Tool Suite**
  - XML schema definition file XML
  - XML stream
  - XML stream

- **Reasoning SDK**
  - XML schema definition file XML
  - XML stream
  - XML stream

- **Boo Labs**
  - XML schema definition file XML
  - XML stream
  - XML stream

- **MXML Toolkit**
  - XML schema definition file XML
  - XML stream
  - XML stream

- **BizTalk**
  - XML schema definition file XML
  - XML stream
  - XML stream

**XML Applications**
- **ASDF+SDF**
  - application
  - application
  - application
  - application

- **Erlang YECC**
  - application
  - application
  - application
  - application

- **JTS Jakarta Tool Suite**
  - application
  - application
  - application
  - application

- **Reasoning SDK**
  - application
  - application
  - application
  - application

- **Boo Labs**
  - application
  - application
  - application
  - application

- **MXML Toolkit**
  - application
  - application
  - application
  - application

- **BizTalk**
  - application
  - application
  - application
  - application

- **TIGRA**
  - application
  - application
  - application
  - application

- **GAML**
  - application
  - application
  - application
  - application
6. Future Work

The first area of future work will evaluate the appropriateness of extended data flow diagrams to abstractly specify ad hoc software interfacing (AHSI) applications. AHSI is concerned with discovering data flow (hyper)paths made by extant procedures, scripts, commands, etc. Choosing an AHSI solution for an EAI application implies that the trade-off analysis that took place beforehand discovered the client’s unwillingness to pay for a unified model of the new application’s data. Therefore AHSI appears to be somewhat at odds with object-orientation (OO). OO modeling is essentially concerned with developing a unified data model from which a family of consistent applications can be derived. In OO class diagrams, associations between classes imply visibility and multiplicity relationships between objects, not data flow. OO class diagram nodes represent classes, the instances of which are computationally active. In an AHSI application such as Figure 6, the diagram nodes represent types or classes, the instances of which may be computationally static or active.

The second area of research will refine the AHSI type system. Currently the AHSI type system assigns each “type” to one of four categories: (memory, predefined access), (memory, ~predefined access), (stream, predefined access) or (stream, ~predefined access). The AHSI type system names all types, even if they represent file types or data structures used in programming languages that do not have named types (e.g., Erlang). Types can be “inherited” by subtypes and be aggregates of other types, but these notions of inheritance and aggregation must be precisely defined. E.g., what is the precise relationship between the type of ANSI C programs and the type of typical C programs that use the OPNET Modeler EMA library? Is the latter so great a restriction over the former that there is no value in specifying a precise relationship between them? Would it be easier to say that the latter is just a complex record type with lots of fancy boilerplate surface syntax?

No doubt the four categories of data can be refined into finer graduations. We will refine the stream/memory type categories into seven gradations according to the International Standards Organization (ISO) Open Systems Interconnection (OSI) 7-layer networking model [30]. Future work will evaluate treating appliponents as protocol stacks and interfacing appliponents as interfacing protocol stacks. The predefined access/~predefined access dichotomy no doubt could be refined into an almost continuous spectrum of accessibility (e.g., 0% accessibility for 100% accessibility). This axis of the type system is also insufficiently objective because a type that is without predefined access for one AHSI application may have adequate predefined access for a different AHSI application.

7. Conclusions

We introduced the world of enterprise application integration (EAI). We introduced the concept of appliponents—software applications that engineers want to use as components in new, larger software applications. We used an informal trade-off analysis to clarify decision parameters for a contrived EAI problem. The trade-off analysis showed that ad hoc, point-to-point data transfer could be the least expensive EAI solution in terms of development costs. We introduced ad hoc software interfacing as a subset of EAI when the option of retrofitting appliponents with middleware is overkill. We asked what could be done to lower the maintenance costs of the ad hoc software interfacing option. We suggested using DSL or XML toolkits to (re)generate ad hoc interfacing software from formal specifications. We noted the overlap between DSL and XML toolkits.

We introduced a framework for specifying ad hoc software interfacing problems. The framework emphasizes reuse of extant data types and extant functions that operate on those types, even when the resulting solution is inelegant. We introduced a four-quadrant space for categorizing extant data types based on whether data occur in memory or streams and whether access to those data is predefined or not. We enumerated the 16 combinations of data transfer between the four quadrants. Each combination outlined the generic data transfer procedure in terms of a few primitive operations. We noted the similarities between our four-quadrant space of data types and the concept of a protocol stack. We introduced a special kind of class diagram to specify ad hoc software interfacing problems and solutions. We discussed Microsoft BizTalk Server as a promising technology for implementing maintainable ad hoc software interfacing solutions. We surveyed a collection of DSL and XML toolkits and applications.
Reyes, Espino, Mohan, Nadkar: Ad Hoc Software Interfacing: Enterprise Application Integration (EAI) when Middleware is Overkill

8. References


[31] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1994.