Software Interoperability through De-facto Standard and Supporting Environment

Nalinee Sophatsathit
Computer Science Program, Faculty of Science and Technology
Suan Sunandha Rajabhat University

Peraphon Sophatsathit
Advanced Virtual and Intelligent Computing (AVIC) Center
Department of Mathematics, Faculty of Science
Chulalongkorn University

Abstract

The proliferation of software application in daily operations is undeniably deep-rooted and intertwined with every fabric of life. Complicated as software systems are, they evolve in much the same manner as the chaotic world of their creator. Yet no one complains about the shortcomings of these heterogeneous software systems created by unassociated developers that were never meant to interoperate. We propose a novel framework that assembles existing techniques to glue together pieces of software. The simplicity of abiding by a de-facto standard and straightforward implementation of a well-established process is the key to realization of the proposed framework. As such, succeeding modification, reorganization, or even re-invention of this novel idea can be systematically performed to establish greater extent of software interoperability environments and pave way for interoperable software systems development.

Keywords: software/tool interoperability, XML, software engineering environments.

1. Software Engineering Environments

Modern software development paradigms focus on commonality of processing format to facilitate interoperability. The good old notion of integrated development environment (IDE) has been rejuvenated as a means for creating, operating, integrating, storing, retrieving, and maintaining of the desired information. Unfortunately, most IDEs operate on homogeneous basis, whereby foreign data must undergo format conversion upon importation, let alone the software itself. In so doing, the ultimate goal of the IDE as being truly interoperable is defeated. Moreover, the extraneous data and format conversion inevitably introduces additional processing burden that, in many cases, does not justify the efforts in exchange for imperfect conversion and information loss. Ironically, software engineering environments in primitive IDE forms have been around since the advent of Ada Programming Support Environment (APSE). APSE and its derivatives, **C**ommon **APSE** **I**nterface **S**et, (CAIS) [1], as well as their counterpart development environment family—the Portable Common Tool Environment (PCTE) and PCTE+ [2], were the two most comprehensive but homogeneous Integrated Programming Support Environments (IPSE) then. Admittedly, they were never fully taken off commercially. More environment specifics, such as P^NMPI inter-tool communication [3], offer dynamic loading and concurrent use at the expense of some infrastructure compliance overhead. At any rate, newer IDEs/IPSEs are more commonplace in development community, e.g., the .NET, EJB, and CORBA, yet still preserve their locality. For all practical purposes, any foreign software or tools must comply with the underlying mandate in order to co-exist.

The expansion of distributed processing also imposes additional interoperability requirements among software systems across heterogeneous platforms. The distribution does not render these software systems to operate at higher level of amalgamated integration. Interoperability is

usually achieved through API, message interfaces, command-line options [4], as well as an information structure model describing the communication bindings between *ToolCommunications* [5] structures. These endeavors exemplify the needs for resolving heterogeneity that may exist among software systems on the same or across platforms. The latter issue is usually handled by means of Workflow Management Coalition [6], as well as meta-data and ontology mechanisms [7] to ensure reliable operation of intercommunicating systems. Fortunately, one predominant effort has established itself as a de-facto standard is the semi-structured XML which permits flexible cross-platform development and software interoperability. This simplicity of textual platform independent framework does not come without a small penalty. It cannot exist alone without a supporting IDE/IPSE to complement its potential. Nevertheless, a more involved problem is the inherent semantic complexity brought about by the underlying language and support systems. Decker, et. al. [8] studied the roles of Document Type Definition (DTD), Resource Description Framework (RDF) schema, and applied Ontology Interchange Language (OIL) to investigate semantic knowledge representation as far as expressive power, syntactic and semantic interoperability are concerned, thereby fostering long-term semantic interoperability.

2. Reference Architecture

As software operations span the distributed computing network, conventional software/tool interoperability through API, message interfaces, and command-line options, serve as a comprehensible and convenient interface. The notion of Component Mill architecture [9] furnishes an infrastructure for component integration on heterogeneous environments by exposing the meta-component model and constructs to supporting technologies. Interoperability enhancement can be further fine-tuned with the help of dynamic and late binding mechanisms as software is executed as a separated application. Unfortunately, these provisions inevitably introduce a new layer of human interconnecting complexity ranging from command language semantic, analysis and design abstractions, user-friendliness overhead, code and style legibility restrictions, and so on.

Bearing the above issues in mind, we propose a novel configuration based on XML technology as a test-bed for software interoperability. The combined knowledge of XML versatility and platform independence makes up a reference architecture of the proposed approach. The critical artifact is a simple and straightforward byte-level encoding scheme that is machine readable, whereby no syntactic or semantic processing overhead is incurred.

The proposed framework encompasses two simple steps that will enable straightforward software interoperation as illustrated in Figure 1. The first step (a) involves creating XML schemas to denote individual software interface artifacts and mechanisms. This has been practiced in many existing applications, particularly database and Internet related work. The second step (b) is to translate all XML schemas by a "*native byte-assembler*" into byte-level instructions in the same manner as the 2-pass assembler. These byte-level instructions, by no means being confined to machine language, are then installed on the target machine programming support environment, thus enabling software systems to co-exist and work together. The simplicity of direct translation from standard XML code to byte-level instructions by passes the aforementioned abstractions, semantic, human-oriented complexities, as well as XML support overhead. As such, it is self-contained, light weight, platform independent, and machine readable that lends itself to machine-to-machine communication without human intervention.
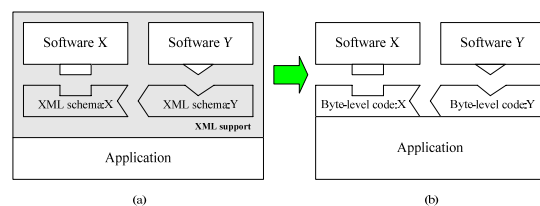

Figure 1 Byte-level instruction of the corresponding XML code.

To demonstrate the above 2-step procedure from an operating standpoint, the participating software configuration is schematically described and represented by XML constructs. This process is the state-of-the-practice in many today's cross-platform development. Procedurally, rather than leaving the output XML code to be further processed by down-stream software, the proposed approach translates XML code into byte-level

instructions as illustrated in Figure 2. The sample XML code segment [10] gets translated into byte-level instructions in the same manner as assembly language translation. The instructions are subsequently executed in the target environment, enabling heterogeneous software modules that were never design to work together to procedurally interoperate. One important pre-translation consideration is organization of the XML source. Some refactoring activities [11] may be called for to ensure that the rearrangement is efficient, non-redundant, and yet still preserves software behavior and quality attributes after translation.

```
<xs:element name="EnrolledCourse">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Grade" type="xs:string"/>
        <xs:any namespace="##any" minOccurs="0"
              maxOccurs="unbounded" processContents="lax"/>
      </xs:sequence>
      <xs:anyAttribute namespace="##any" processContents="lax"/>
    </xs:complexType>
</xs:element>
```

```
xs_element         csect                                      ;begin level 1
label1_1           BYTE          "EnrolledCourse"
xs_complexType     csect                                      ;begin level 2
xs_sequence        csect                                      ;begin level 3
label3_1           BYTE          "Name"
label3_2           BYTE          "Grade"
xs_any             begin_blk
namespace3_1       BYTE          "##any"
minOccurs          DEC           0
maxOccurs          DEC           Nan                          ;unbounded
processContents    BYTE          "lax"
                   end_blk                                    ;xs_any
                   end_csect                                  ;xs_sequence
xs_anyAttribute    begin_blk
namespace2_1       BYTE          "##any"
processContents    BYTE          "lax"
                   end_blk                                    ;xs_anyAttribute
                   end_csect                                  ;xs_complexType
                   end_csect                                  ;xs_element
```

Figure 2 Byte-level instruction of the corresponding XML code.

The rationale behind the proposed byte-level instruction is two folds. First and foremost, byte-level instructions permit easy and speedy machine execution of various software interchanges as they are native code and there are virtually no format and data conversions required. Second, byte-level instructions are portable on any target machines. As software is ported, the corresponding XML schemas are cross-compiled through the target machine *native byte-assembler* during system configuration. Subsequent modifications, plug-and-play set ups, and a variety of IDE/IPSE supports can be incorporated or reconfigured accordingly with minimal efficiency and performance tradeoffs. Such a small and machine processible representation makes this framework ideal for applications running on limited resource devices such as embedded code, mobile agent interconnection protocols, foreground/background processing, and heterogeneous software interoperability exchange. The only price tag is building the desired portable *native byte-assembler*

to be installed on any IDE/IPSE host where byte-level instructions or other forms of native code can be generated for the designated target machine.

3. Case studies

The viability of the proposed approach is demonstrated in two case studies contrived to illustrate the principles. The first case was a small and simple spreadsheet created by a popular software and exported (in CSV format carrying no extra conversion overhead) to be displayed on a mobile phone, running its own software that was never meant to work with the spreadsheet software. Figure 3 depicts the handcrafted byte-level instructions. However, the conversion was carried out manually due to technical difficulties inhibited by the phone's capability. Eventually, these byte-level instructions were further translated with the help of J2ME support, a well known IDE/IPSE for mobile phone application tool. In so doing, it re-affirmed the notion of flexible interoperability without confining to native machine instruction or platform-specific mandate. Nonetheless, the experiment was never thoroughly tested over the actual mobile network owing to legal commission rights of local phone operators and communication law prohibition of individual access on public frequencies. At any rate, this over-simplified case study was devised to demonstrate how the proposed schema could be realized with available state-of-the-practice technologies.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<project name="score" default="csv" basedir=".">
   <tally name="show">
       <property name="stock_id" value="xs:string"
                 count="0" price="0"/>
   </tally>
</project>
```

```
xml             prog
version         BYTE          1.0
encoding        BYTE          "UTF-8"
                end_prog
project         START
name            BYTE          ""
default         BYTE          "csv"
basedir         BYTE          "."
tally           csect
name            BYTE          "show"
property        begin_blk
name            BYTE          "stock_id"
count           DEC           0
price           DEC           0
                end_blk                              ;property
                end_csect                            ;tally
                END                                  ;project
```

Figure 3 Byte-level instruction of the header.

The second case study demonstrates the importance of IDE/IPSE's role in cross platform development. It was a fun-filled mobile phone game called "*Mobigocchi*" that was created as a senior project from the same J2ME support. The application was subsequently downloaded to selected models of mobile phone equipped with compatible IDE/IPSE. Principal components of the game are depicted in Figure 4. It was primarily played on a stand-alone mobile phone, or against another mobile phone player over Bluetooth protocol. The game won the third prize in the 2007 Collegiate Mobile Game Development competition organized by a local phone operator.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<project name="" default="jar" basedir=".">
<description>Builds, tests, and runs the project .</description>
<import file="nbproject/build-impl.xml" />
<!-- Written by: Wizarut Niyomsart and Maytita Charoenrat      -->
<!-- Dept of Mathematics, Faculty of Science, Chulalongkorn University -->
<!--
   BattleCanvas.jav           background templates
   BattleClient.jav a         client
   BattleSever.java           server
   Godji.java                 game main function
   MobiBluetooth.java         Bluetooth connection
   MobiCanvas.java            phone display background
   MobiNormal.java            phone interface
-->
</project>
```

Figure 4 Principal components of a mobile phone game.

One important issue concerning this novel approach is commercialization. As XML is a de-facto open standard that has been widely accepted by both academia and industry, the *native byte-assembler* can be made proprietary based on specific platforms and needs. Better yet, third party developers are free to abolish this *native byte-assembler* and opt for their own proprietary implementation such as the use of J2ME support in this article. Consequently, interoperable software systems development can still be realized without sacrificing commercial leverage and trade secrets.

4. Conclusion

This article proposes a novel byte-level framework that complements normal use of XML versatility to permit interoperable software systems development. The notion of IDE/IPSE supports is fully exploited to enhance software/tool interoperability. As developers create myriad of software systems for general or specific purposes, there bounds to be new requirements precipitating from various software applications that call for the software systems to co-exist and interoperate. Without attempting to do it all, the proposed approach introduces a 2-step process based on well-practiced disciplines that is simple and straightforward to implement on any platform. Optionally, XML code refactoring may be needed to optimize schema organization. The overall provisions entail greater software interoperability that not only augments the-state-of-the-practice interoperable software systems development, but also broadens the horizon of machine learning

research and development. It is hope that the efforts expended by the software community to reckon with the silver bullet will arrive at applicable robust mechanisms akin to what tangible goods already possess [12].

5. References

[1]   **C**ommon **A**PSE **I**nterface **S**et, *MIL-STD-1838A*, September 30, 1989.

[2]   ECMA Technical Committee (TC33)—ECMA TR/55, A Reference Model for Frameworks of Computer-Assisted Software Engineering Environments, European Computer Manufacturers Association (ECMA), 114 Rue du Rhone - CH - 1204 Geneva (Switzerland), December 1990.

[3]   Schulz, M.; de Supinski, B.R, "A Flexible and Dynamic Infrastructure for MPI Tool Interoperability", *International Conference on Parallel Processing 2006 (ICPP 2006)*, August 2006, pp. 193-202.

[4]   Yimin Bao and Ellis Horowitz, "A New Approach to Software Tool Interoperability", *Proceedings of the 1996 ACM symposium on Applied Computing (SAC '96)*, pp. 500-509.

[5]   Harvey, J.G.; Marlin, C.D, "A Layered Operational Model for Describing Inter-tool Communication in Tool Integration Frameworks", *Proceedings of 1996 Australian Software Engineering Conference,* 14-18 July 1996, pp. 55-63.

[6]   Hazem T. El-Khatib, M. Howard Williams, David h. Marwick, and Lachlan M.

Mackinnon, "Using a Distributed Approach to Retrieve and Integrate Information from Heterogeneous Distributed Databases", *The Computer Journal*, Vol. 45, No. 4, 2002, pp. 381-394.

[7]   Ngamnij Arch-int and Peraphon Sophatsathit, "A semantic information gathering approach for heterogeneous information sources on WWW", *Journal of Information Science*, Vol. 29, No. 5, 2003, pp. 357-374.

[8]   Stefan Decker, Sergey Melnik, Frank Van Harmelen, Dieter Fensel, Michael Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks, "The Semantic Web: The Roles of XML and RDF", *IEEE Internet Computing*, Vol. 4, Issue 5, September-October 2000, pp. 63-74.

[9]   Ly Danielle Sauer, Robert L. Clay, and Rob Armstrong, "Meta-Component Architecture for Software Interoperability", *Proceedings of the International Conference on Software Methods and Tools 2000 (SMT 2000)*, 6-9 Nov. 2000, pp. 75-84.

[10] "Service Modeling Language, Version 1.1", W3C Working Draft 6 August 2007, http://www.w3.org/TR/2007/WD-sml-20070806/

[11] Tom Mens and Tom Tourwe, "A Survey of Software Refactoring", *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, February 2004, pp. 126-139.

[12] Brad Cox, "No Silver Bullet Revisited", *American Programmer Journal*, 1995.