An Executable Literate Document (ELD) for Application System Development

Yu-Liang Chi[1] and Chien-Hua Tsai[2]
[1]Department of Management Information System
Chung Yuan Christian University, Taiwan, R.O.C.
maxchi@cycu.edu .tw
[2]Department of Management Information System
National Defense University, Taiwan, R.O.C.
A334589@ms48.hinet.net

Abstract

Formal software development relies upon software requirement specifications (SRS) to translate customer requirements to final products. Literate documents are responsible for a communication medium among readers in different development phases. However, issues of document inaccuracy and inconsistency are crucial problems. This paper provides some perspectives of how SRS utilized executable literate document (ELD) can be directly linked in software development process to improve throughput, reduce iteration, and avoid uncertainty. The idea of this study divides application system into two parts, ELD and necessary programming code. ELD is an XML-based document that contains business rules retrieved from original application software system. Since ELD is human readable document, user can via change ELD to easily modify application system. Therefore, ELD is not only software requirement document but also a semi-application.

Keywords: user-centered, executable document, XML

1 The Problem

Software applications are artificial results from human. For years, there were many theories, specs, standards, and etc. published to help software applications development. Most of them are based on developer-centered concepts that focus on how to speed up developing time, clarify user requirements, reuse exist components, and so on. One of the most important purposes of these assisting tools is reducing gaps between software developers and end-user. Because developers and users belong two different domains, it is no reason to ignore end-user role in software development [6]. This paper describes an approach, called executable literature document (ELD), to involving the participation of end-user who potentially play a key role in the whole cycle of software development stages. Using such an approach allows us to execute specifications directly and improve accuracy and consistency of applications.

The first goal of ELD is to execute specifications directly. In previous studies, formal specification languages (FSL) allowed executing specifications directly via a mathematical notation used in software development to express the functional specification of a system [1][5][7]. The well-known FSL are popularly used in the world, such as VDM-SL and Z notations [2][3]. However, FSL are still a developer-centered approach to solve problems. ELD and FSL are different in 1) ELD is based on text document and FSL is based on language-like document. 2) ELD is designed for both developers and end-users, but FSL is designed only for developers. 3) ELD is a part of an application that retrieved only business rules into a standalone and modifiable document. The similar researches can be found on US Department Of Defense (DOD) Information Systems Agency (DISA) project [4]. DISA runs an XML registry for the Defense Information Infrastructure (DII) at Common Operating Environment (COE). The major benefit of COE developers is making reusable common data components and related specifications available to all potential requesters. Also, business rules isolated from systems are maintainable without original system designer.

The second goal of ELD is to keeping accuracy and consistency during each phases of application

development. In software development life cycle, software requirement specification (SRS) is an early and critical stage to collect, transforms, and deploys knowledge for follow-on user [9]. SRS is describing the requirements of a computer system from the user's point of view. In general speaking, an SRS document specifies the required behaviors and attributes of a system. For decades, requirement definition and requirements specification were usually written by natural language. The issues of them are inaccuracy in translating to continue phase of software development life cycle. Although formal specification language, model-based specification, algebraic specification, and etc. improve and shorten the conversion of each phases. However, they are hardly understood by regular end-users. In the other hand, ELD is a text-based document that structured by markup language. End-users capably participate in development life cycle not only in SRS, but also in later maintenance. Since the structured properties of ELD document, it also can act as same as an executable specifications. The similar research can be found on Guerrieri's article [10]. Guerrieri also utilized XML as an medium to carry software document, but not emphasize business rules.

2 Introduction of ELD

2.1 What is an ELD

ELD, Executable literature document, introduces a new concept and utilization of software development. In our design, ELD broke up hierarchical architecture of software development process; instead, ELD was tight integral into software development process. It has literate-rendering characteristics as usual and may extend to contain the executable business rules, open interface, and version control for whole life cycle of software development process.

In Figure 1, the relationship between ELD and other software processes is presented. Since the potential capability of markup mechanism is obvious, various kinds of document such as MRD, SRD, SD and SA also can be written by markup language. Therefore, ELD can inherent and reuse the other kind of software document. Since ELD shared the responsibility of business logics, the system implementation will be relative simple. Also, the unit and system testing can be leveraged by different error of logic. ELD also can expand to contain the open interface; therefore, system will transparent for system or enterprise integration process. Currently, ELD was concentrate on the utilization of software design specs and system implementation.

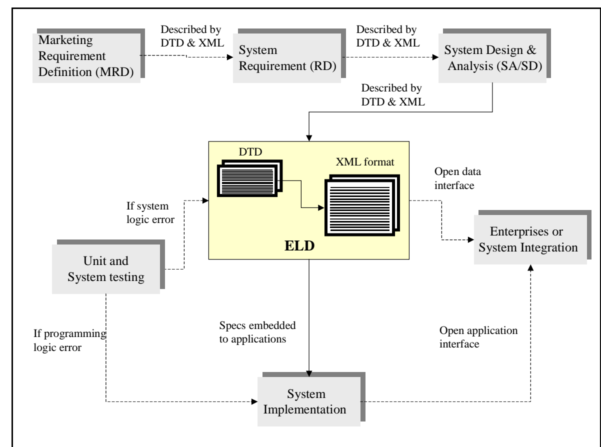

Figure 1: The architecture of ELD in software development life cycle

Machines can't really read, but they can interpret literature document. Markup mechanism simply aids this interpretation. For enforcing machines to understand literate document, ELD applied markup mechanism by utilizing XML that is created by the World Wide Web Consortium to complement the weakness of HTML. XML promises an internet-based, universal standard data format, and human/machine readable markup language. Data Type Definition (DTD) is a set of rules for document construction. DTD rules the syntax for XML document, such as what tag can use in document, what order they would appear in, which tags can appear inside other ones.

ELD utilizes both XML and DTD to well-formed literate software document. Moreover, XML processor aids on reading and interpreting document into machines. XML processor loads the documents and related DTD files, checks it follows all rules, and builds a document tree structure that can be processed by applications. Therefore, a machine-readable document was generated.

2.2 ELD is an Executable Document

Based on machine-readable fundamental, an executable document still needs the management mechanisms to navigate and process XML formatted document. Since the XML processor can parse the ELD document that is also a tree-like structure, Document Object Model (DOM) was naturally utilized to deal with this tree object [11]0. In this research, we utilize DOM to address the document tree-structure issues. The DOM had an object design that operates document as a tree-like structure. A basic DOM object is called node that may or may not contain its child nodes. DOM is an application-programming interface (API) for markup document to enhance how to build, navigate, and

process the contents of document.

In Figure 2, general requirements and business logic are written on markup-formatted document. XML parser then parsed markup document into machine. In an implementation stage, for examples, programming file does not duplicate the business logic anymore. Instead, the software document can be directly imported and DOM can help to access the right actions. Since the relationship of software document and implementation is tight integral, it reduces the ambiguity of specs transformation and also improves the productivity movement of software process.

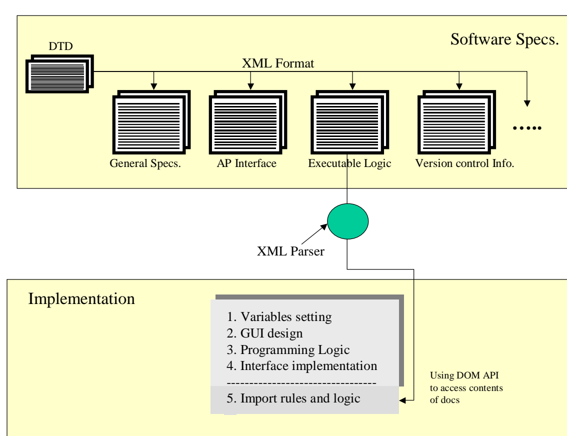

Figure 2: How ELD can be executed

3 How to Implement ELD on Software Development

For quickly understanding the idea of ELD, we assume the software requirements well defined already and then started from design software document. A simple control system, alarm clock, is used to demonstrate what the ELD look like. In Figure 3, we use Finite state machine (FSM) to address the problem domain. It includes three states: “Reset, Alarm, and Idle”, and two events: “TimeUp and Switch”. These states and events mix up twelve different control situations that later fire a propriety action. The literate software requirement document of alarm clock control system can be written as List 1.

3.1 A n DTD for Alarm Clock Control System

Data Type Definition (DTD) is an optional procedure for creating XML document [11]. In formal procedure, a DTD is used to define the elements that may be used and dictates where they may be applied in relation to each other DTD also help to verify the contents of XML files and to avoid the errors. The DTD of an alarm clock control system can be defined as List 2. In this List, DTD is clearly defined the hierarchy of each element. For examples, the document will start from “ALARMCONTROL” that includes five sub elements: TITLE, STATE, INPUTSIGNAL, EVENTLIST, and IDL. DID is also clearly described the relationship; for examples, “EVENT” is parent of “SWITCH” and “TIMEUP”.

List 2 was stored on a file named “alarmclock.dtd” that will be refereed by XML file later. DTD file can be flexibly located either on local or Internet.

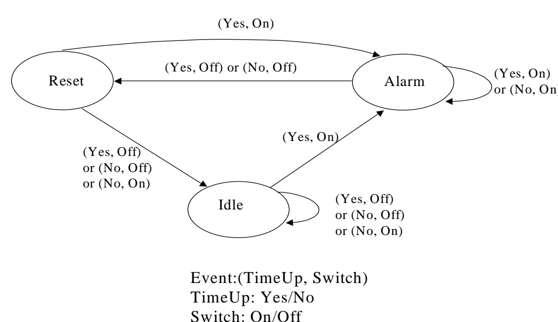

Figure 3: Alarm clock control system using FSM

3.2 An DTD for Alarm Clock Control System

In this section, we follow the DTD of alarm clock control system that defined on previous section to create software document. This research did not interesting on how XML editor operates; however, several commercial XML editors were available in markets. Also, develop a special purpose of XML editor for editing software document is possible.

For convenience, we separate the List 3, alarm clock control system using ELD with two parts. The upper part is descriptions of requirements that are not related business logics. The lower part is about business logics of alarm clock. Again, XML file also as same as DTD file can be located in anywhere. Except the markup tags, the contents and meaning in List 3 is totally as same as List 1. In front of List 3, a DTD file, “alarmclock.dtd”, was imported to verify the format between DTD and XML. In the lower part of List 3, an EVENTLIST node contained the all business logic of alarm clock. Twelve EVENTs are embedded between a start tag, <EVENTLIST>, and end tag, </EVENTLIST>.

The literate software document was been created now. An XML parser then used to parse the content of document into a machine. XML parser API or tool is for decoding markup tag and available in various programming languages.

Alarm Clock Control System Specs

1. Alarm clock control system has three states. There are:
   - Reset
   - Alarm
   - Idle
2. Alarm clock control system has two input signals. There are:
   - Timeup: Yes, No
   - Switch: On , Off

Each state will have four events, which consist of a pair of input signals, respectively. The signature of each event is: (Timeup, Switch). Therefore:

Current State	*Event*		*Action*
	TimeUp	*Switch*	
Reset	*Yes*	*On*	*Alarm*
Reset	*Yes*	*Off*	*Idle*
Reset	*No*	*On*	*Idle*
Reset	*No*	*Off*	*Idle*
Alarm	*Yes*	*On*	*Alarm*
Alarm	*Yes*	*Off*	*Reset*
Alarm	*No*	*On*	*Alarm*
Alarm	*No*	*Off*	*Reset*
Idle	*Yes*	*On*	*Alarm*
Idle	*Yes*	*Off*	*Idle*
Idle	*No*	*On*	*Idle*
Idle	*No*	*Off*	*Idle*

List 1: Alarm Clock control system design spec

```
<!ELEMENT ALARMCONTROL (TITLE?,STATE, INPUTSIGNAL, IDL. EVENTLIST)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT STATE (DESCRIPTION?,STATEITEM+)>
<!ELEMENT DESCRIPTION (#PCDATA)>
<!ELEMENT STATEITEM (#PCDATA)>
<!ELEMENT INPUTSIGNAL (DESCRIPTION?,SIGNAL+)>
<!ELEMENT SIGNAL (NAME?,TYPE+)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT TYPE (#PCDATA)>
<!ELEMENT IDL (DESCRIPTION?,NAME, STATEMENT+)>
<!ELEMENT STATEMENT (#PCDATA)>
<!ELEMENT EVENTLIST (EVENT+)>
<!ELEMENT EVENT (CURRENTSTATE,CURRENTEVENT+,ACTION)>
<!ELEMENT CURRENTSTATE (#PCDATA)>
<!ELEMENT CURRENTEVENT (TIMEUP, SWITCH)>
<!ELEMENT TIMEUP (#PCDATA)>
<!ELEMENT SWITCH (#PCDATA)>
<!ELEMENT ACTION (#PCDATA)>
```

List 2: An DTD for Alarm Clock control system

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ALARM CLOCK CONTROL SYSTEM “alarmclock.dtd”>
<ALARMCONTROL>
<TITLE>Alarm Clock Control Requirement Specification </TITLE>
<STATE>
        <DESCRIPTION>Alarm clock control system has three states. </DESCRIPTION>
        <STATEITEM>Reset</STATEITEM>
        <STATEITEM>Idle</STATEITEM>
        <STATEITEM>Alarm</STATEITEM>
</STATE>
<INPUTSIGNAL>
          <DESCRIPTION>Alarm clock control system has two input signals. </DESCRIPTION>
   <SIGNAL>
          <NAME>TimeUp</NAME>
                 <TYPE>Yes</TYPE>
          <TYPE>N0</TYPE>
          </SIGNAL>
   <SIGNAL>
          <NAME>Switch</NAME>
          <TYPE>On</TYPE>
          <TYPE>Off</TYPE>
            </SIGNAL>
</INPUTSIGNAL>
  <EVENTLIST>
     <DESCRIPTION> Each state will have four events, which consist of a pair of input
     signals, respectively. The signature of each event is: (TimeUp, Switch).
     </DESCRIPTION>
     <EVENT>
   <CURRENTSTATE>Reset</CURRENTSTATE>
   <CURRENTEVENT>
          <TIMEUP>Yes</TIMEUP>
          <SWITCH>On</SWITCH>
             </CURRENTEVENT>
   <ACTION>Alarm</ACTION>
     </EVENT>
     <EVENT>
          <CURRENTSTATE>Reset</CURRENTSTATE>
     <CURRENTEVENT>
          <TIMEUP>Yes</TIMEUP>
          <SWITCH>Off</SWITCH>
     </CURRENTEVENT>
          <ACTION>Idle</ACTION>
     </EVENT>
                                  :
                                  :
     <EVENT>
          <CURRENTSTATE>Idle</CURRENTSTATE>
     <CURRENTEVENT>
```

List 3: An ELD for Alarm Clock control system

3.3 How to Execute the ELD via Programming

After the XML formatted literate software document imported, we using DOM to navigate the document. The whole document is consists of several levels of nodes. In Figure 4, alarm clock control system software document (ELD) can be drawn as a tree-like structure. The first level node is a document root, ALARMCLOCK that contains four child nodes. Each child nodes may or may not have its child nodes. Node is the fundamental technical of DOM. There are several different kind nodes such as element node, text node, and so on. In Figure 4, for example, "CURRENTSTATE" is an element node tag name and "Reset" is text node value.

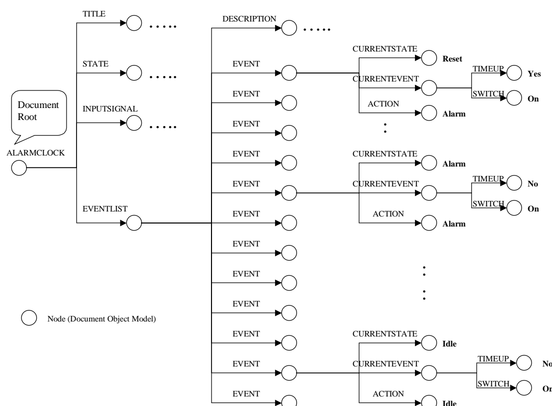

Figure 4: Object Model of Alarm clock control system

Since literate software document deals with executable business logic, it simplifies the programming logic of implementation. In List 5, Java programming language examples, it contains necessary variables such as the location of software document, states, time, and switch. The programming functions part can be divided to six parts as following:

1. Show the software document: parse the markup document (List 3) to literate rendering document (List 1).
2. Parse software document to object document. It also navigates and finds specific business logic. The result is stored the twelve EVENT nodes into a vector and then return the vector.
3. Sorting the vector using the criteria "CURRENTSTATE" and its value. The result is stored the four EVENT nodes into a vector and then return the vector.
4. Sorting the vector using the criteria "TIMEUP" and its value. The result is stored the two EVENT nodes into a vector and then return the vector.
5. Sorting the vector using the criteria "SWITCH" and its value. The result is stored the one EVENT nodes into a vector and then return the vector.
6. Output the final result and print out the propriety action.

```
import java.io.*;
import java.util.*; // for store node
public class alarm_clock {
   public static void main(String argv []) {
String location="file:c:/clock.xml";
  String cevent_v = argv[0];
   String timeup_v = argv[1];
   String setsw_v    = argv[2];
   Vector content, ss, vv, vv1,vv2;
   FindNode FN, FN1;
  FindVector FV1,FV2,FV0;
  writeout WO;

//1. Show the software document
WO= new writeout(location);
  content= (Vector) WO.pop();
  WO.pop_node_value(content);

//2. Parse software document, Select business logic:Event =>12 nodes
FN= new FindNode(location, "EVENT");
  ss= (Vector) FN.find_T();

//3. Sorting vector, criteria CURRENTSTATE and value =>4 nodes
FV0= new FindVector(ss,"CURRENTSTATE",cevent_v);
vv = (Vector)FV0.find_V();

//4. Sorting vector, criteria TIMEUP and value =>2 nodes
        FV1= new FindVector(vv,"TIMEUP", timeup_v);
  vv1 = (Vector)FV1.find_VP();

//5. Sorting vector, criteria SWITCH and value =>1 nodes (Final)
FV2= new FindVector(vv1,"SWITCH", setsw_v);
  vv2 = (Vector)FV2.find_VP();

//6. Output Final Result
String fire_action= (String) FV2.store_single(vv2,"ACTION");
System.out.println("The Action will be ==>" +fire_action);
}}
```

List 4: An Implementation of Alarm Clock control system using Java

List 4 has two reusable utility classes, FindNode and FindVector, which we wrote for processing

literate business logic. If the customer requirements are change, developers simply revised the contents of document without the effort in the other process. If the control system becomes very complex, for examples one hundred switches, it also easily be done by reusing utility classes. Since the literate software document share business logic that is also executable, it is possible to reduce many unnecessary iterations of software development process.

4 Conclusion

This paper introduces a new design of literate software document that is also machined readable and executable document. The major contribution in this paper emphasized that executable literate document (ELD) retrieved business rules from traditional application development. We utilize markup and DOM mechanisms to process literate document; therefore, an ELD become possible. This design addresses inconsistency and inaccuracy problems in life cycle of software development. ELD also has many potential advantages for software testing, integration, and maintenance. Consequently, ELD is expected to address the ambiguous issues among transforming specifications and reduce unnecessary iterations during software process life cycle.

References

[1] Jonathan Bowen. Formal Specification and Document using Z: A Case Study Approach. *International Thomson Computer Press,* 1996.

[2] Jim Davies and Jim Woodcock. Using Z: Specification, Refinement and Proof. *Prentice Hall International Series in Computer Science*, 1996.

[3] ISO/IEC, Information technology - Programming languages, their environments and system software interfaces - Z notation , *ISO/IEC FCD 13586,* 2001

[4] Dept. of Defense (DoD). Common Operating Environment. *http://diicoe.disa.mil/coe/*

[5] Nancy Leveson. Completeness in Formal Specification Language Design for Process-Control Systems. *Proceedings of Formal Methods in Software Practice Conference*, 2000.

[6] Nancy Leveson. Intent Specifications: An Approach to Building Human-Centered. *IEEE Trans. On Software Engineering*, 2000.

[7] NancyLeveson, Mats Heimdahl, Holly Hildreth, and Jon Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 1994

[8] Nancy Leveson, Heimdahl, M.P.E., Reese, J.D. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. *ACM/Sigsoft Foundations of Software Engineering /European Software Engineering Confernce, Touluse,* 1999.

[9] Kotonya, G. and Sommerville, I. The Requirements Engineering: Processes and Techniques, *John Wiley & Sons, Chichester, England,* 1998.

[10] Guerrieri E.., “Software Document Reuse with XML,” *In P. Devandu and J. Poulin, editors, Pro. 5 th Int. Conf. on Software Reuse,* 1998.

[11] Laurent, S., *XML: A Primer*, MIS: Press, Foster City, CA, 1997.